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

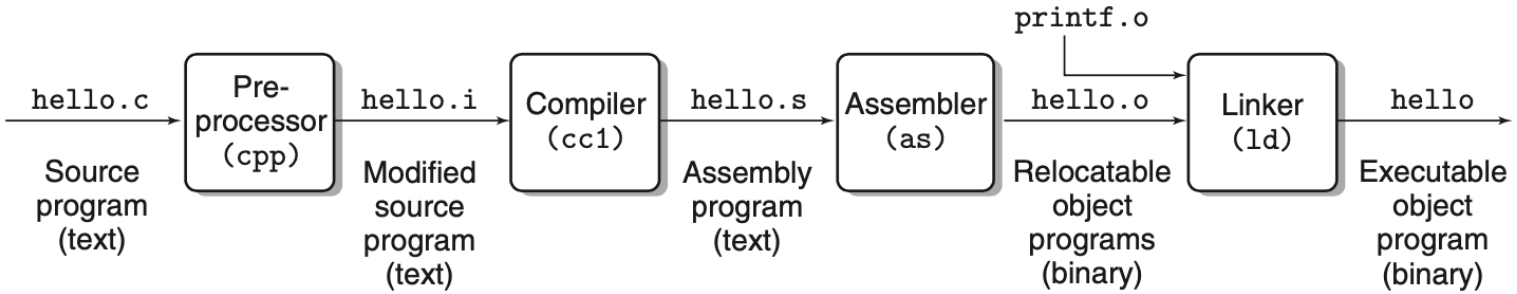


Figure 1.3 **The compilation system.**

Recall

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

GCC

↓ Compiler

- Assembly language (.s)

↓ Assembler

- Machine Code (.o, .so, .a)

↓ Linker

CLANG (LLVM)

↓ Compiler

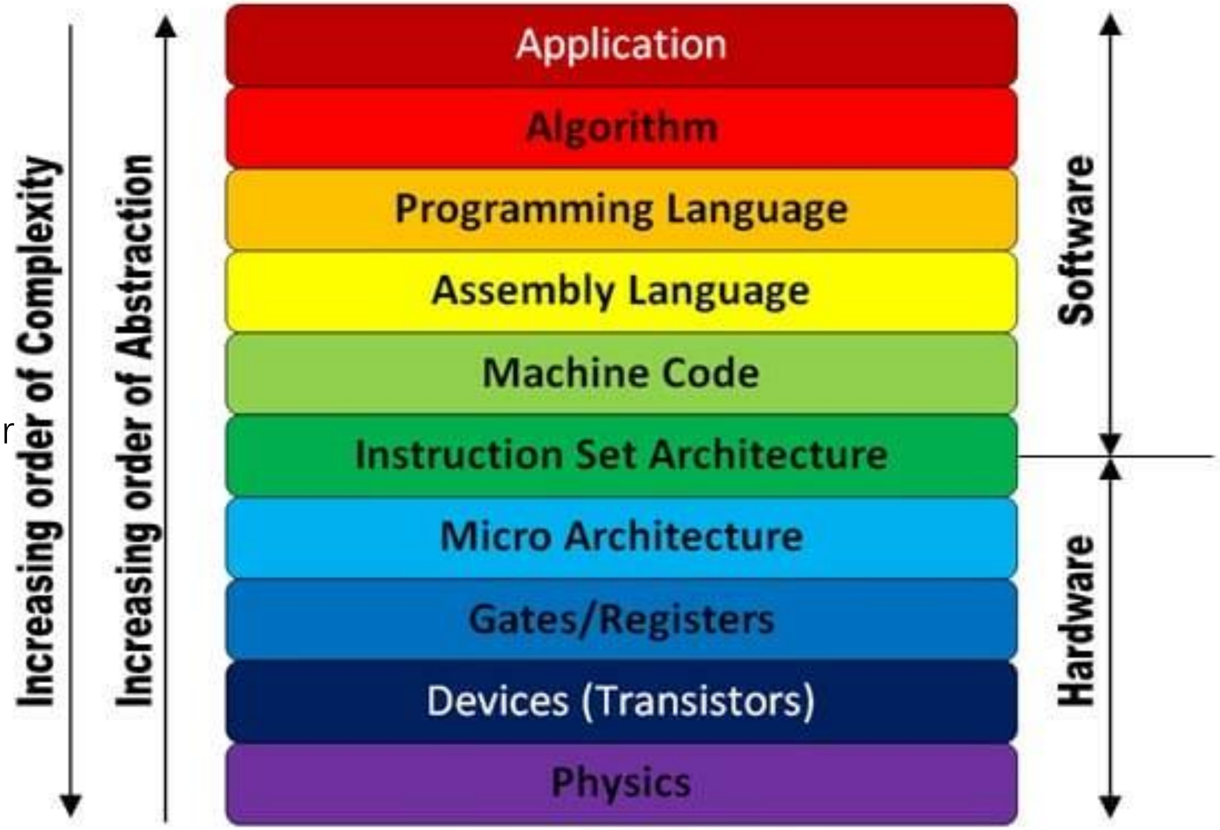
- LLVM IR

↓ LLVM Optimizer

- LLVM Backend

↓ Assembler

↓ Linker



Layers of Abstraction

- 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)

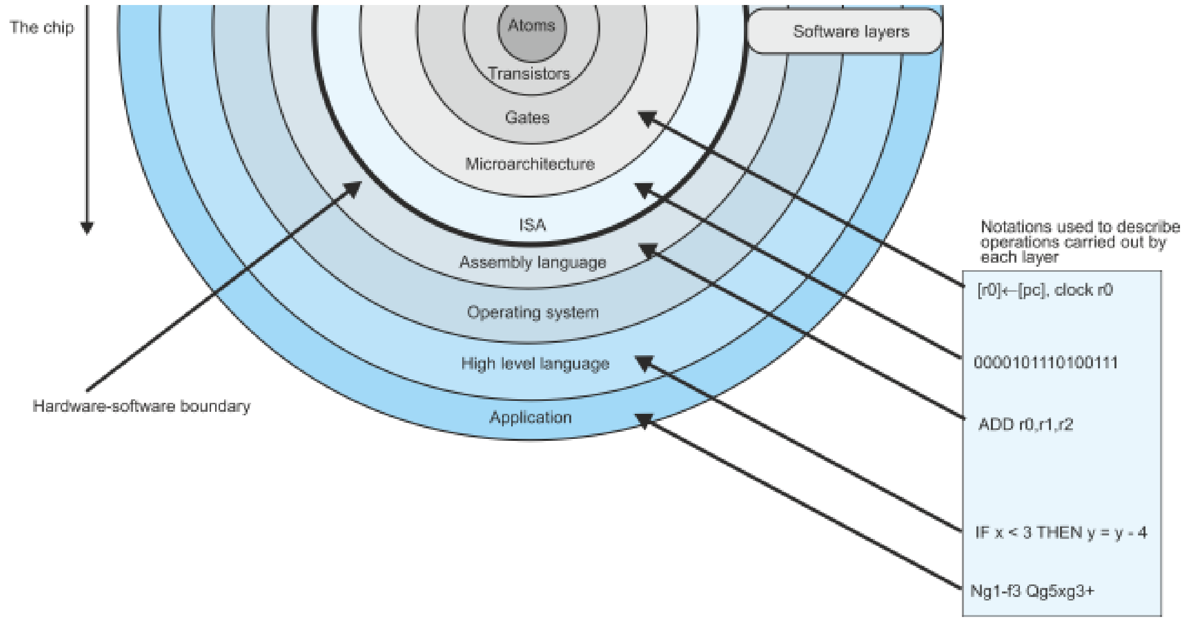
↓ Linker

- Machine Code

Two abstractions: ISA

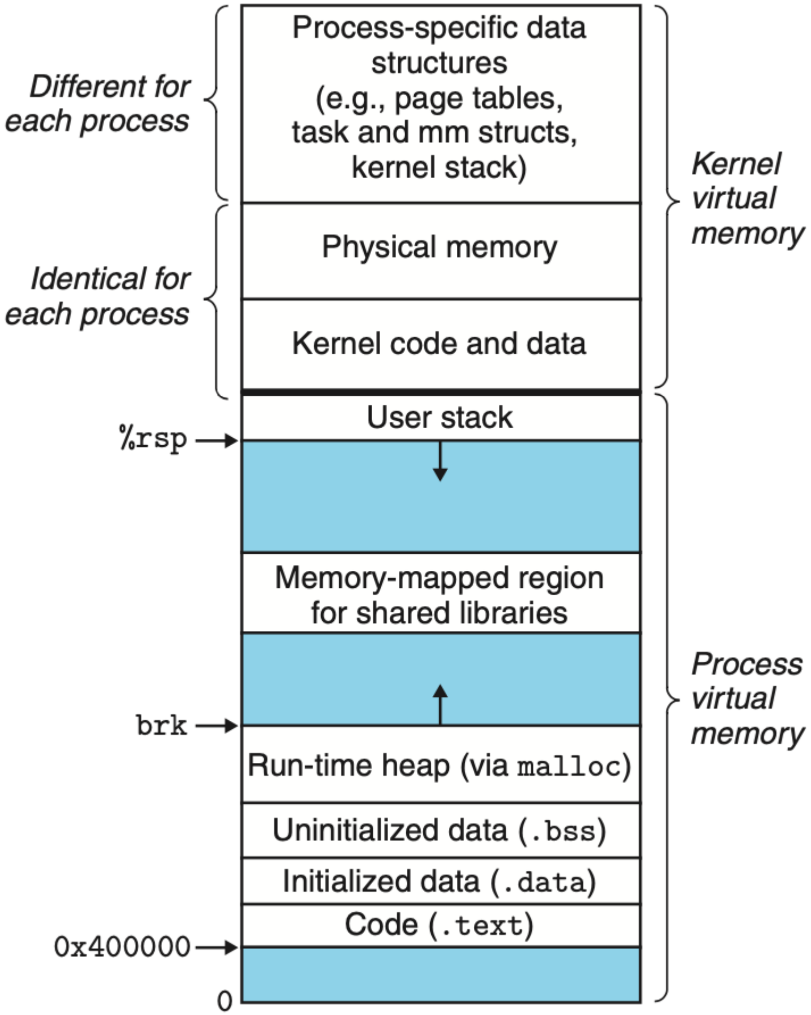
- 定义处理器状态、指令格式和指令对状态的影响
- The hardware/software **interface**
- Examples
  - IA32, AMD64 (x86-64), ARM64, RISC-V, etc.

- ISA ≠ 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


Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%rsp
Memory-mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0x400000
0
Process virtual memory

工具链

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

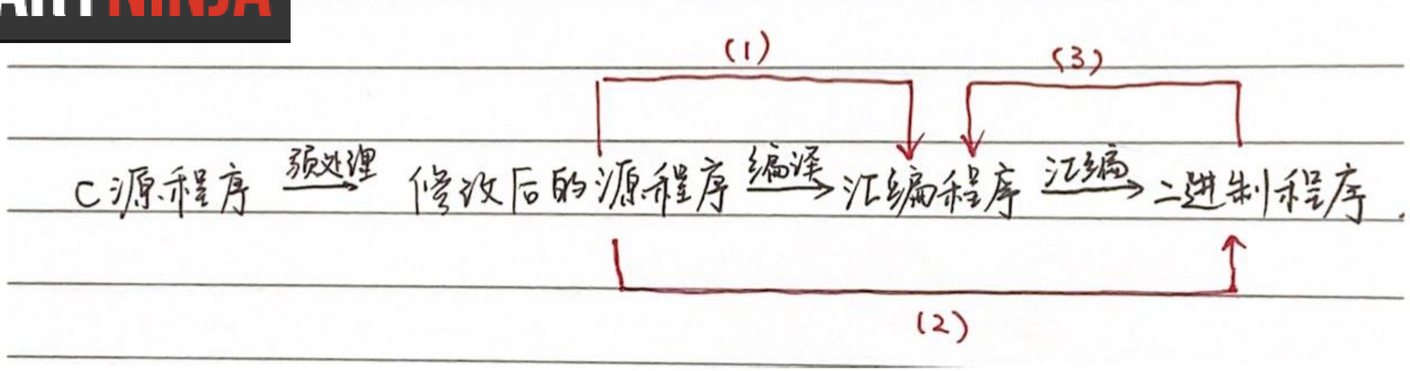
(2) gcc [-O] -c <file> [-o]

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

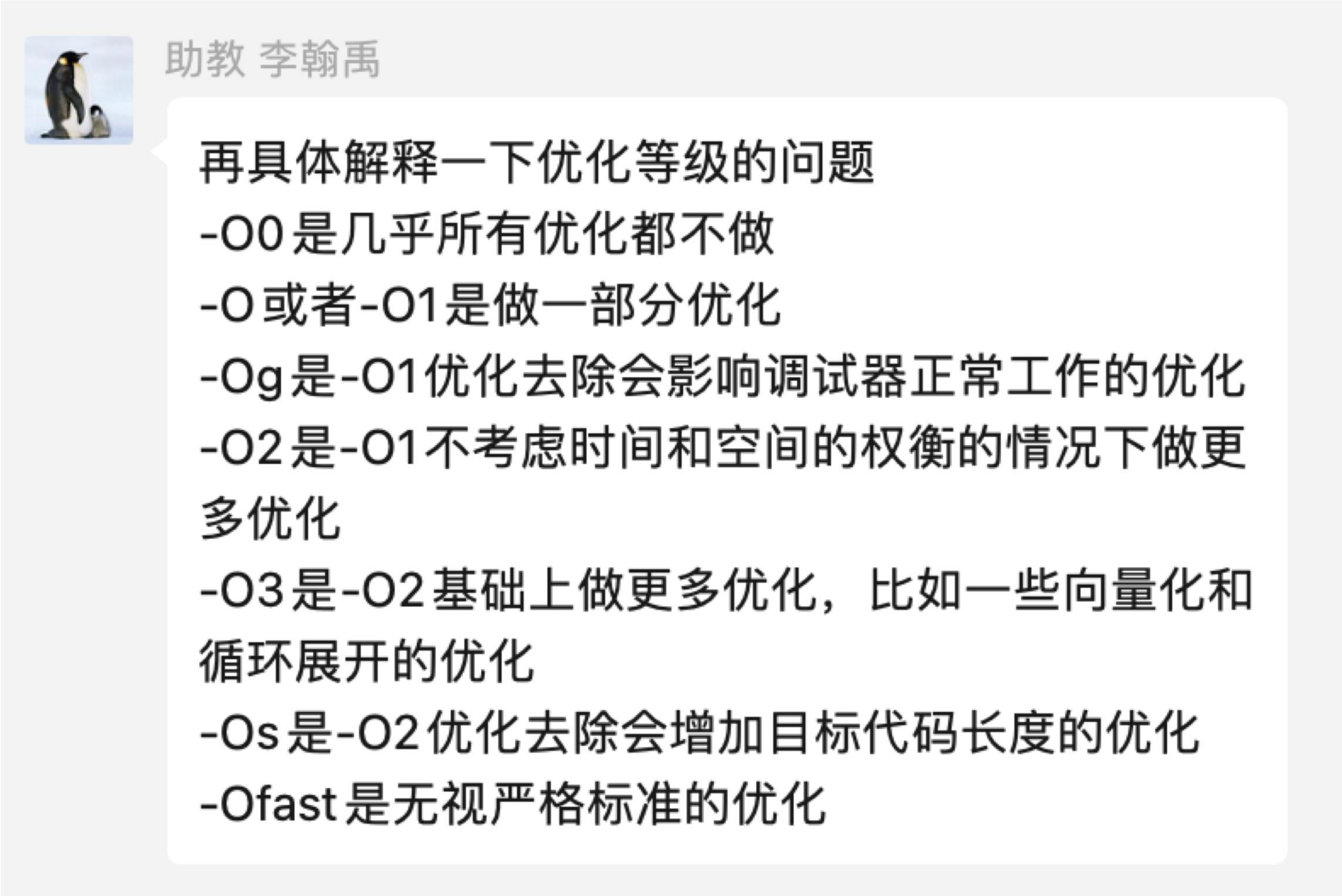
- gdb and bomblab

(c->s) Compiler Explorer COMPILER EXPLORER

(o->c) ByteNinja BINARYNINJA



编译器优化



助教 李翰禹

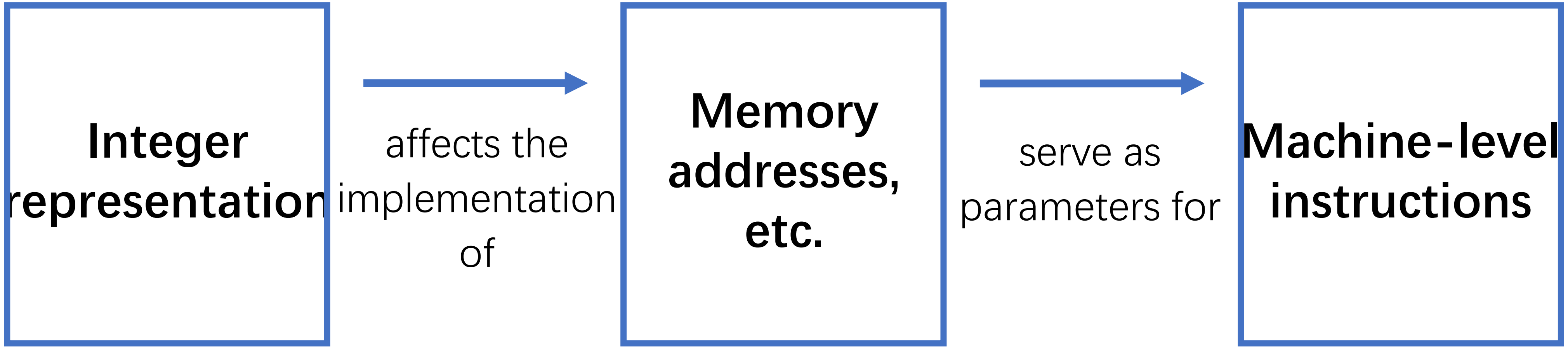
再具体解释一下优化等级的问题
-O0是几乎所有优化都不做
-O或者-O1是做一部分优化
-Og是-O1优化去除会影响调试器正常工作的优化
-O2是-O1不考虑时间和空间的权衡的情况下做更多优化
-O3是-O2基础上做更多优化，比如一些向量化和循环展开的优化
-Os是-O2优化去除会增加目标代码长度的优化
-Ofast是无视严格标准的优化

Basics (CS:APP Ch. 3.1 - 3.5)

贾博暄

Relationships between topics

- (courtesy of TA)



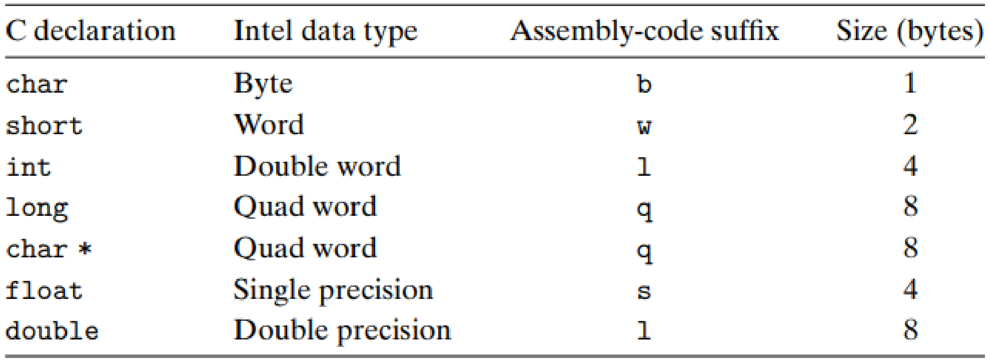
Integer representation affects the implementation of **Memory addresses, etc.** serve as parameters for **Machine-level instructions**

Registers

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

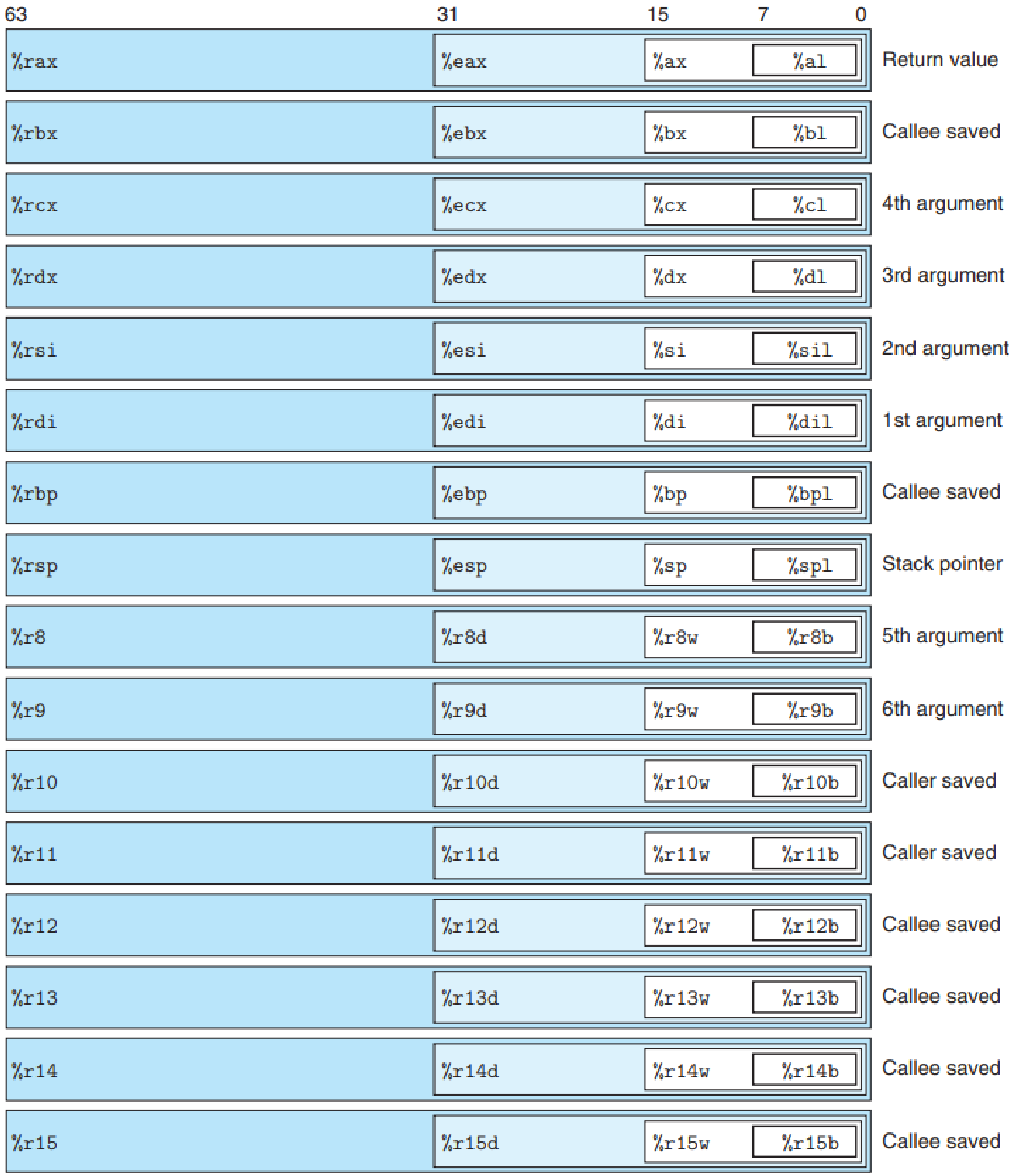
Note on naming

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

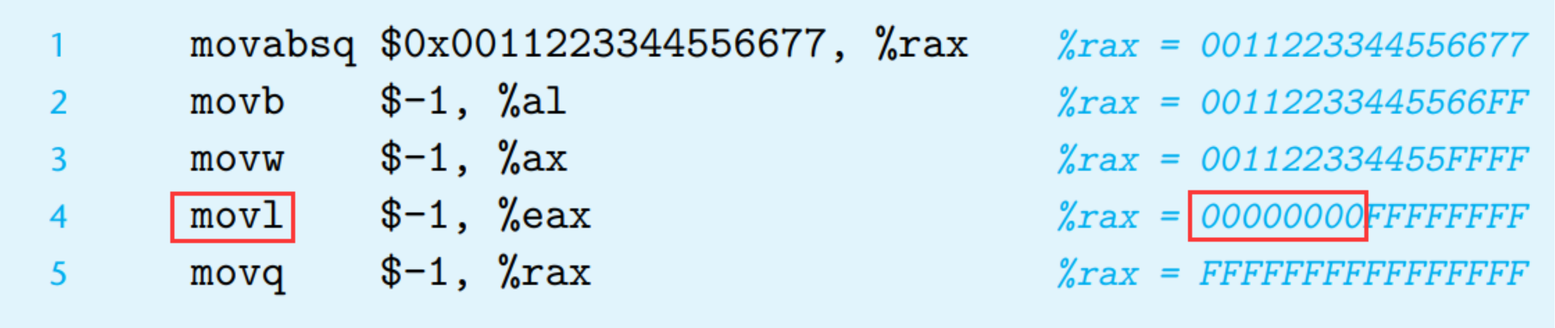
Figure 3.1 **Sizes of C data types in x86-64.** With a 64-bit machine, pointers are 8 bytes long.



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



```
movabsq $0x0011223344556677, %rax     %rax = 0011223344556677
movb    $-1, %al                      %rax = 00112233445566FF
movw    $-1, %ax                      %rax = 001122334455FFFF
movl    $-1, %eax                     %rax = 00000000FFFFFFFF
movq    $-1, %rax                     %rax = FFFFFFFFFFFFFFFF
```

- Consequences
  - **MOV** instructions get a little weird (e.g. there is no **`movzlq`**)
  - 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

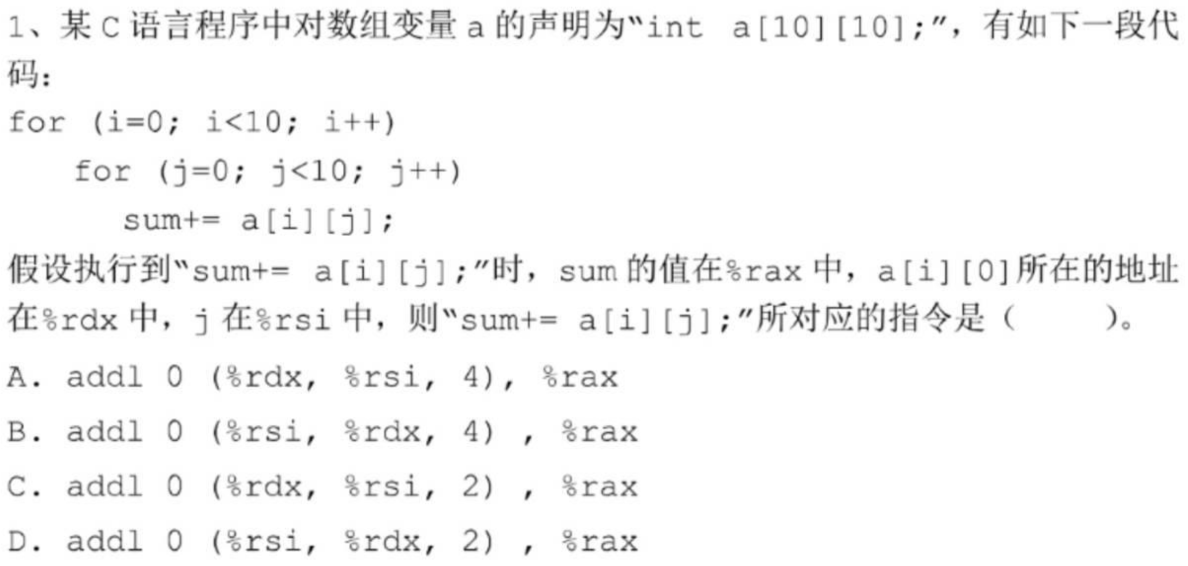
Conventions…

- 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

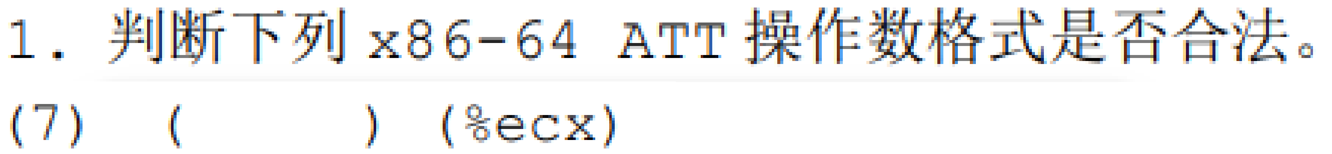
B. addl 0 (%rsi, %rdx, 4) , %rax

C. addl 0 (%rdx, %rsi, 2) , %rax

D. addl 0 (%rsi, %rdx, 2) , %rax

Conventions…

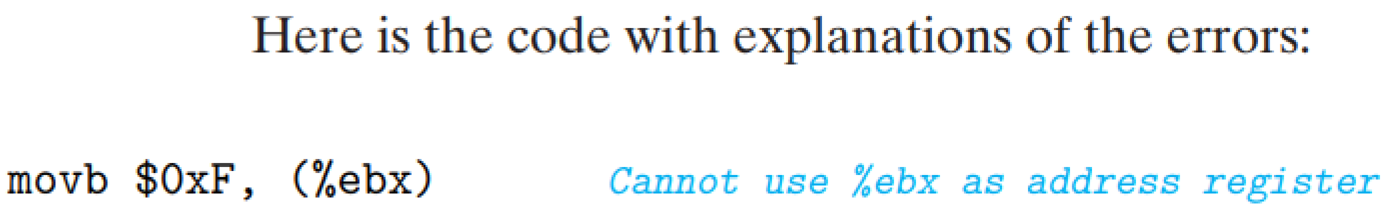
- x86-64 不允许将除 64 位寄存器以外的寄存器作为寻址模式基地址



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

From\To	Imm	Reg	Mem
Imm		√	√
Reg		√	√
Mem		√	

- Memory reference
  - Full form:

$$\text{Offset (Base, Index, scale factor)}$$

$$Imm + \mathrm{R}[rb] + \mathrm{R}[ri] \cdot s$$

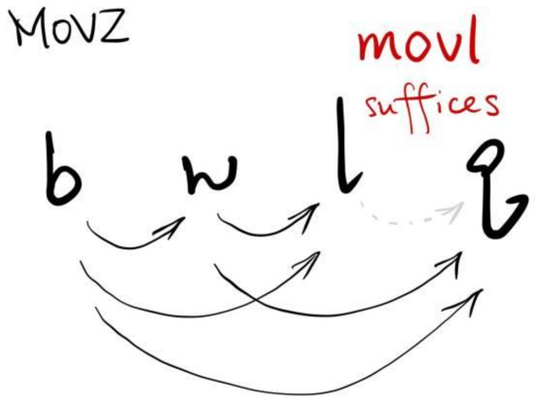
  - Selected omission yields the other formats
  - s limited to 1, 2, 4, 8
  - No $ before Imm here

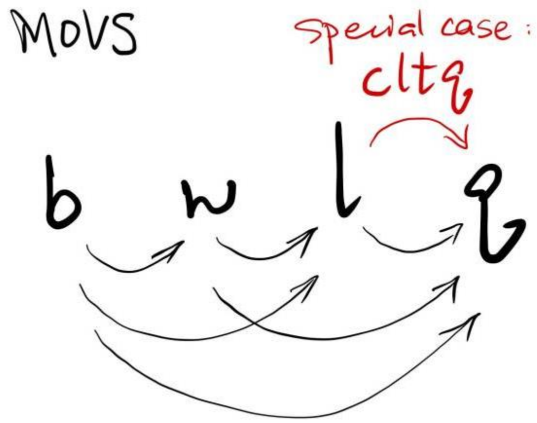
Instruction Classes - Move

- **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 `movzlq`
  - **MOVS**
    - `cltq`
      - effectively a compact encoding of
        `movslq %eax, %rax`



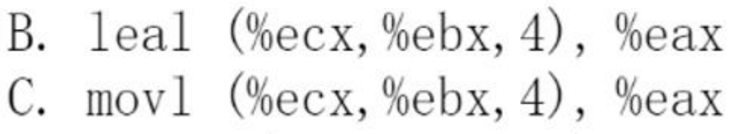


z: zero-extend

s: sign-extend

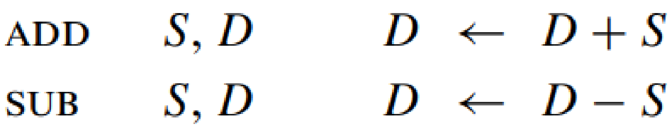
Instruction Classes - Arithmetic

- **`leaq`**
  - (Important!) **`lea`** vs **`mov`**
  - Same format
  - **`lea`** 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	$D \leftarrow D + S$
SUB	S, D	$D \leftarrow D - S$

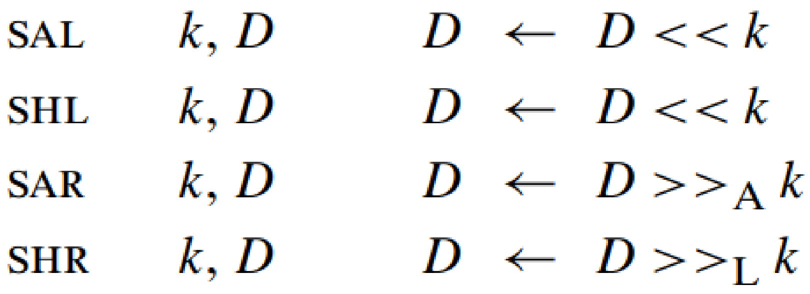
Instruction Classes - Shifts

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

`sh`: logical shift

`sa`: arithmetic shift

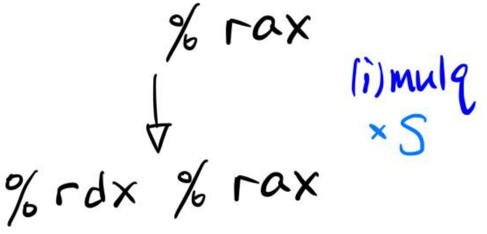
- 1st operand (k) = shift amount
  - **must be $Imm or `%cl`**
  - actual shift amount (also) depends on type of value to be shifted

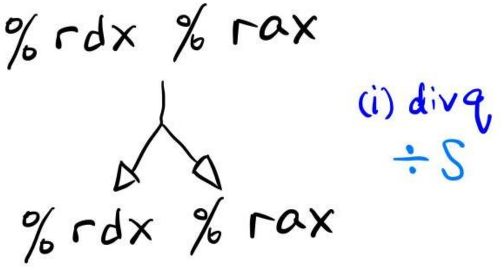


SAL	k, D	$D \leftarrow D << k$
SHL	k, D	$D \leftarrow D << k$
SAR	k, D	$D \leftarrow D >>_A k$
SHR	k, D	$D \leftarrow D >>_L k$

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

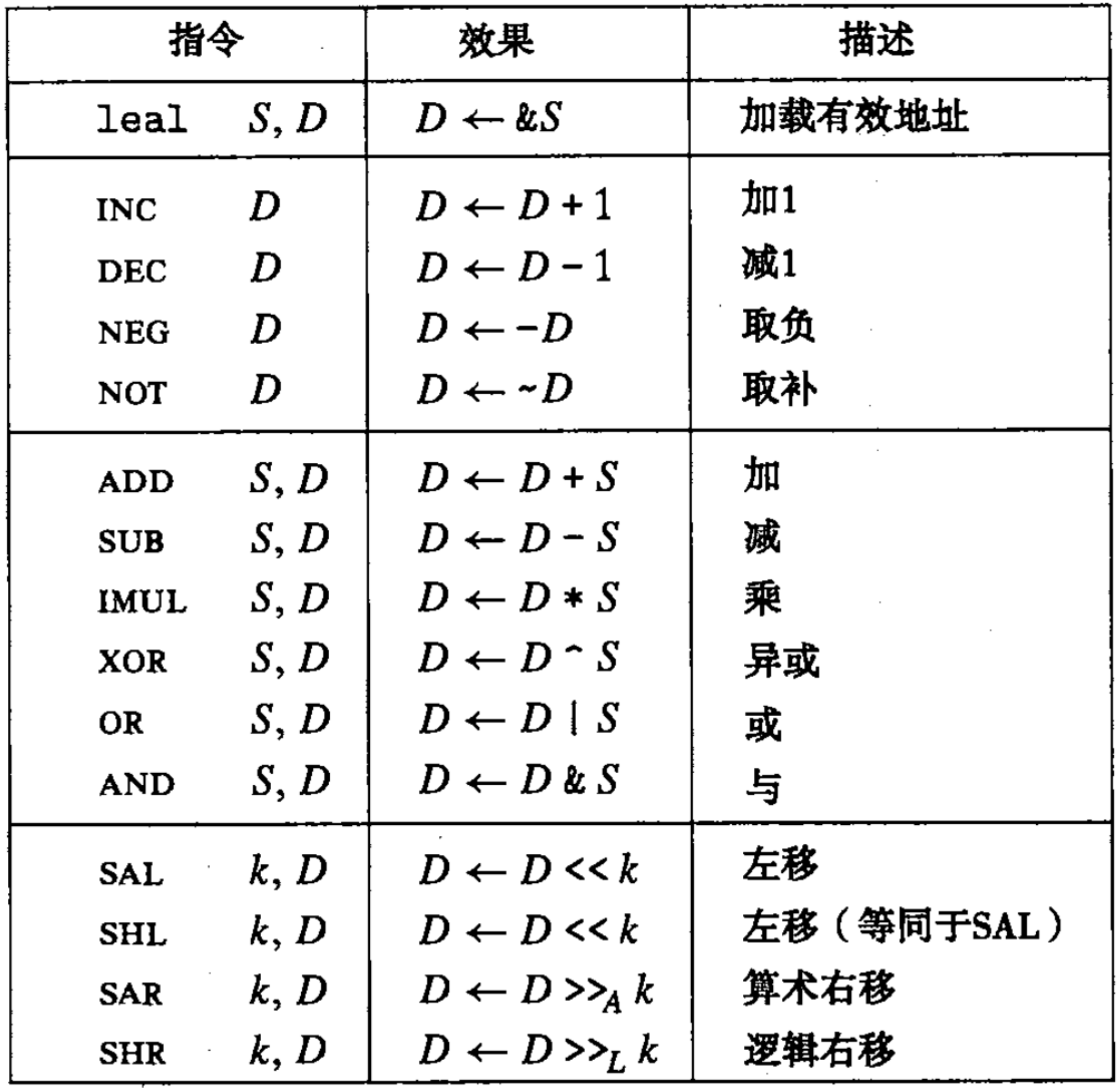
条件码

- 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	$D \leftarrow D$ ^ S	异或
OR	S, D	$D \leftarrow D \mid S$	或
AND	S, D	$D \leftarrow D \& S$	与
SAL	k, D	$D \leftarrow D << k$	左移
SHL	k, D	$D \leftarrow D << k$	左移（等同于SAL）
SAR	k, D	$D \leftarrow D >>_A k$	算术右移
SHR	k, D	$D \leftarrow D >>_L k$	逻辑右移

	CF	ZF	SF	OF
LEA	✗	✗	✗	✗
INC	✗	✓	✓	✓
DEC	✗	✓	✓	✓
NEG	✓	✓	✓	✓
NOT	✗	✗	✗	✗
ADD	✓	✓	✓	✓
SUB	✓	✓	✓	✓
IMUL	✓	✓	✓	✓
XOR	0	✓	✓	0
OR	0	✓	✓	0
AND	0	✓	✓	0
SAL	✓	✓	✓	0
SHL	✓	✓	✓	0
SAR	✓	✓	✓	0
SHR	✓	✓	✓	0

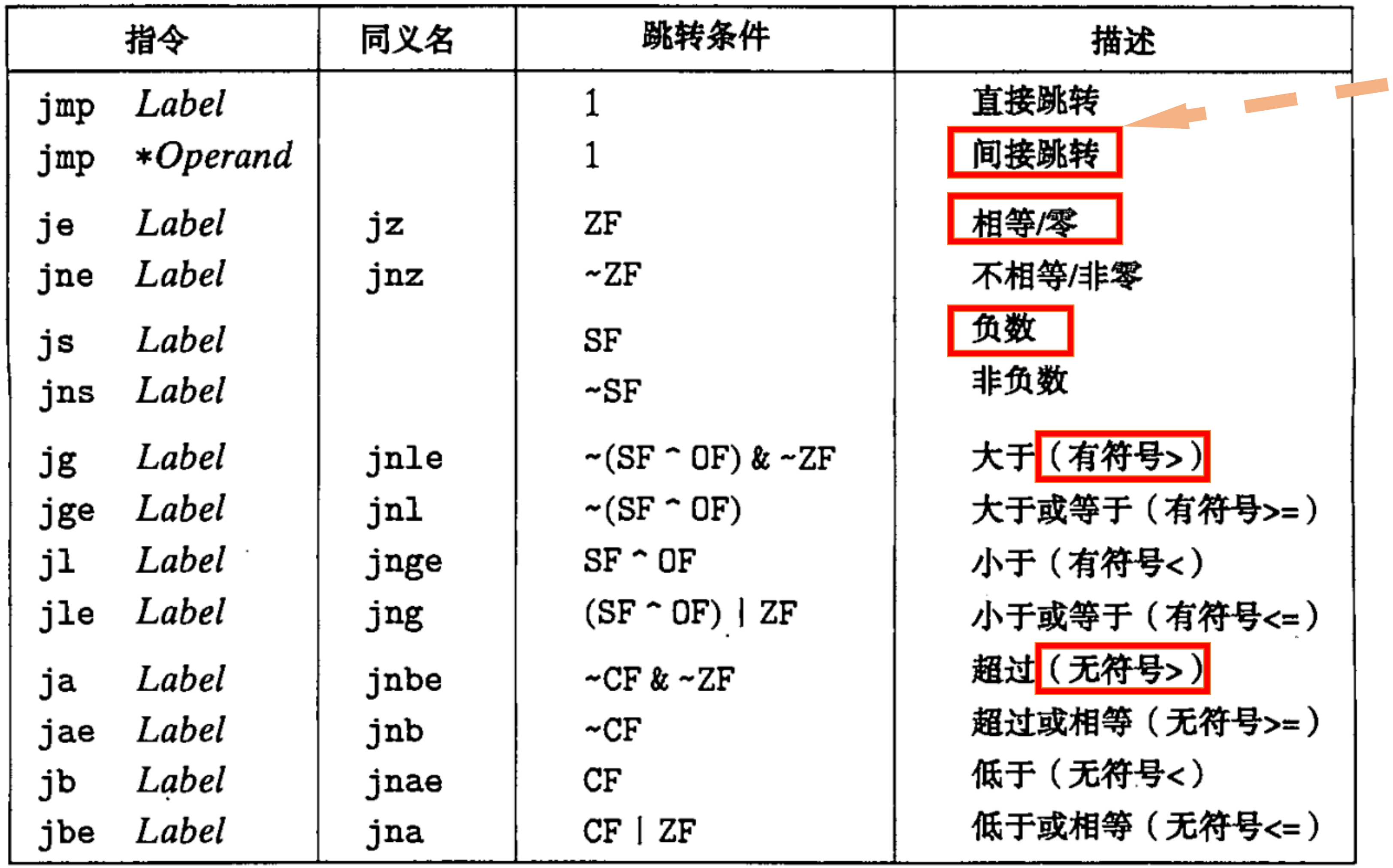
INC, DEC

算术运算

逻辑运算

移位运算

跳转指令



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

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

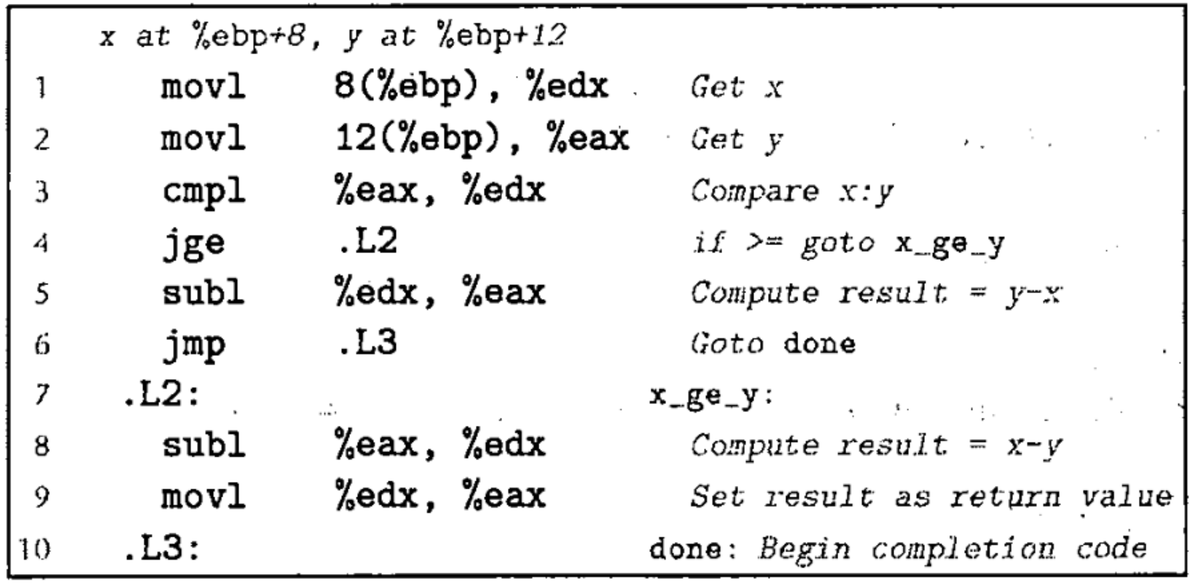
if Not Below

“*” 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

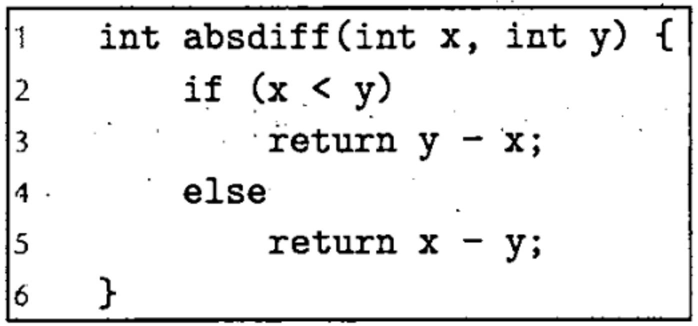
逆向工程和goto

- 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
      movl    8(%ebp), %edx       Get x
      movl    12(%ebp), %eax      Get y
      cmpl    %eax, %edx          Compare x:y
      jge     .L2                 if >= goto x_ge_y
      subl    %edx, %eax          Compute result = y-x
      jmp     .L3                 Goto done
    .L2:                        x_ge_y:
      subl    %eax, %edx          Compute result = x-y
      movl    %edx, %eax          Set result as return value
    .L3:                        done: Begin completion code
```

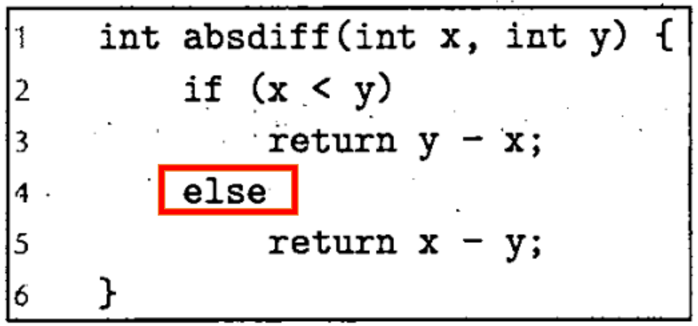
c）产生的汇编代码



```
int absdiff(int x, int y) {
    if (x < y)
        return y - x;
    else
        return x - y;
}
```

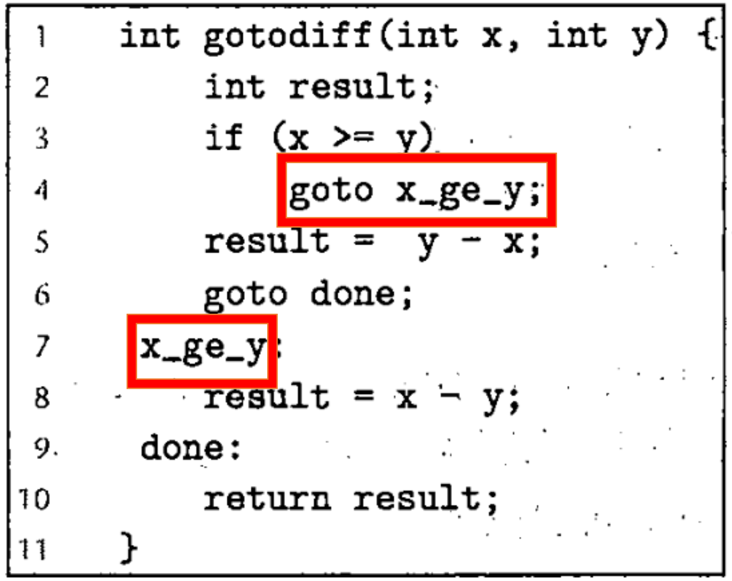
a）原始的C语言代码

goto：正向视角



```
int absdiff(int x, int y) {
    if (x < y)
        return y - x;
    else
        return x - y;
}
```

a）原始的 C 语言代码

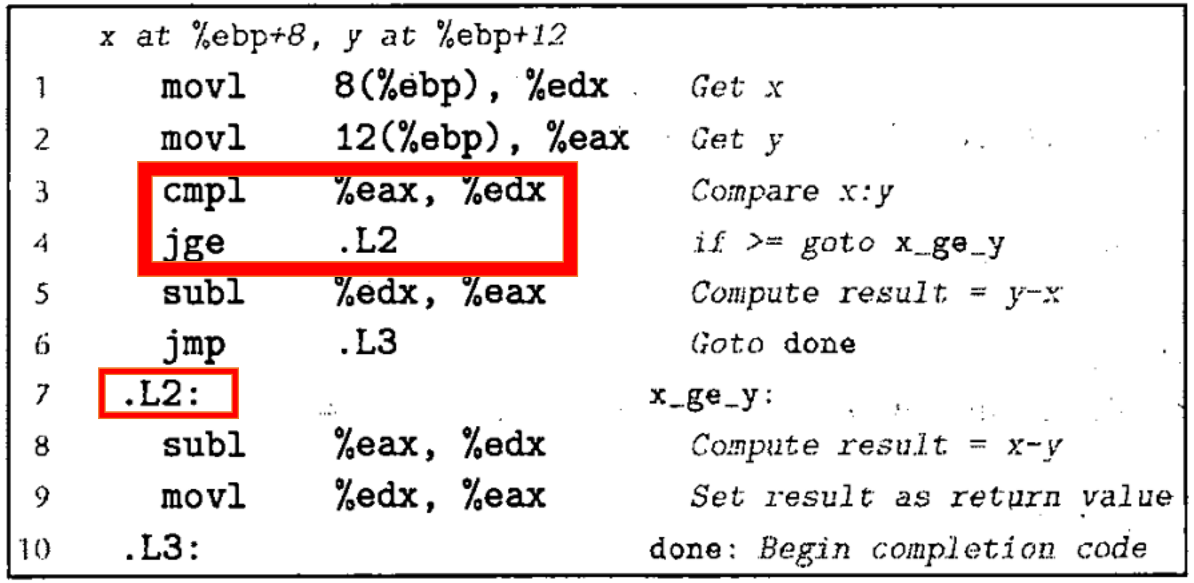


```
int gotodiff(int x, int y) {
    int result;
    if (x >= y)
        goto x_ge_y;
    result =  y - x;
    goto done;
 x_ge_y:
    result = x - y;
 done:
    return result;
}
```

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…

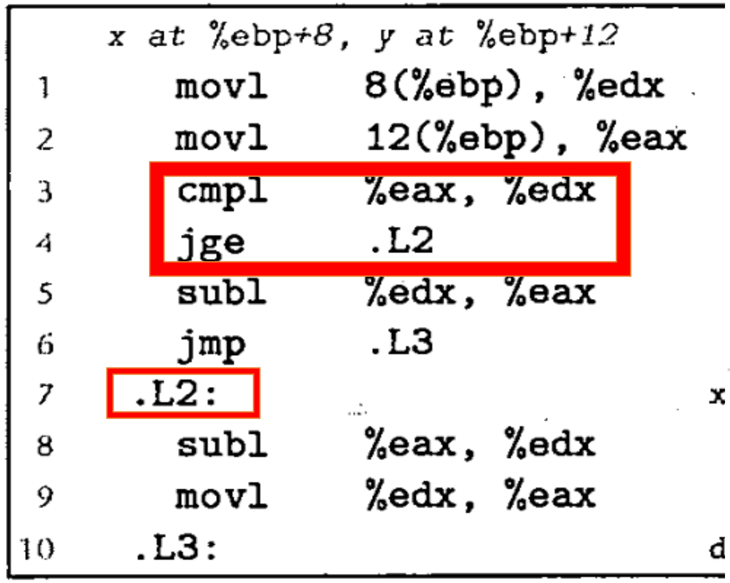
如何把else转化为汇编?



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

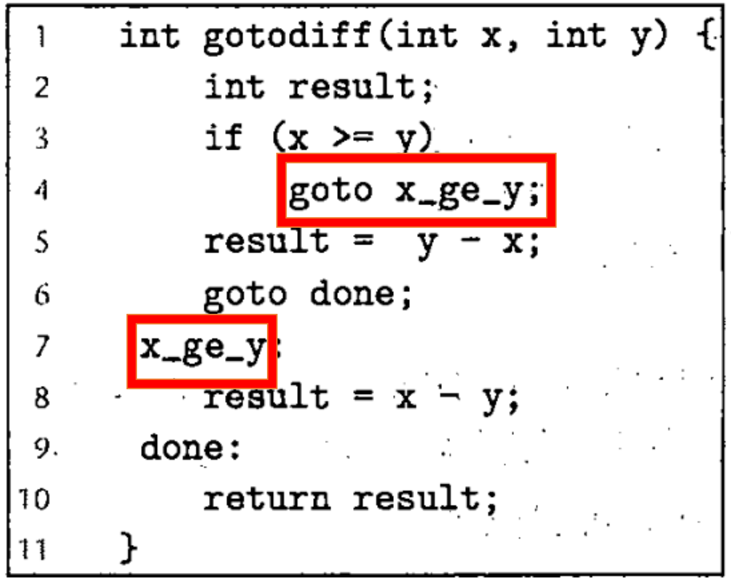
c）产生的汇编代码

goto：逆向视角



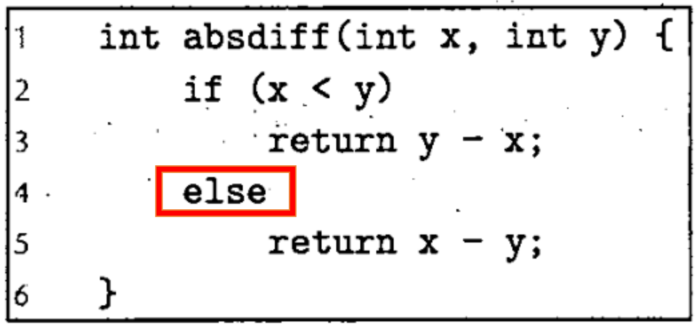
```
    x at %ebp+8, y at %ebp+12
      movl      8(%ebp), %edx
      movl      12(%ebp), %eax
      cmpl      %eax, %edx
      jge       .L2
      subl      %edx, %eax
      jmp       .L3
   .L2:                                x
      subl      %eax, %edx
      movl      %edx, %eax
   .L3:                                d
```

c）产生的汇编



```
int gotodiff(int x, int y) {
    int result;
    if (x >= y)
        goto x_ge_y;
    result = y - x;
    goto done;
 x_ge_y:
    result = x - y;
 done:
    return result;
}
```

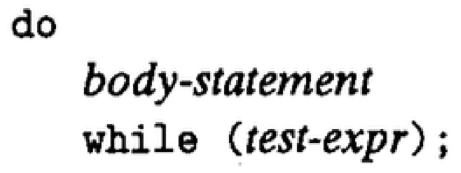
b）与之等价的 goto 版本



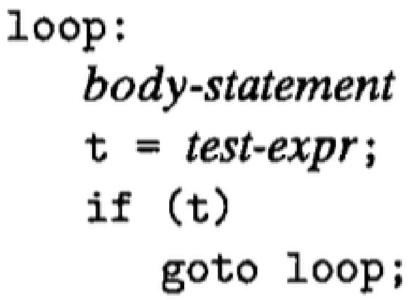
```
int absdiff(int x, int y) {
    if (x < y)
        return y - x;
    else
        return x - y;
}
```

a）原始的 C 语言代码

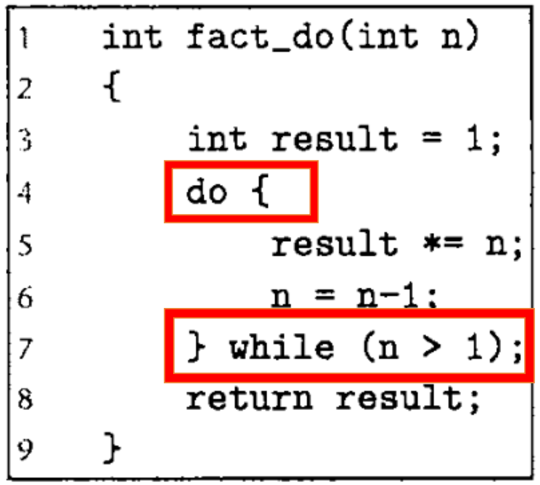
循环：do-while



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

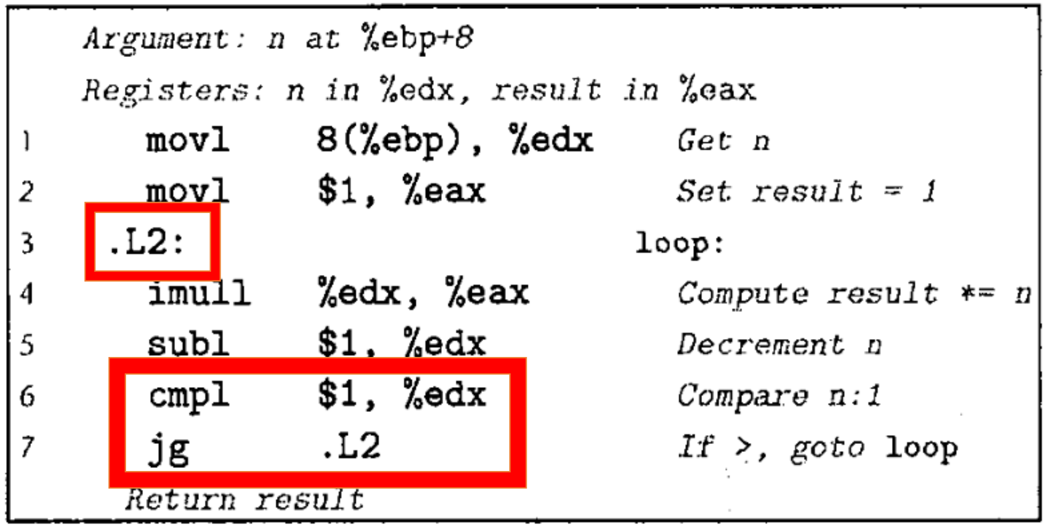


```
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
```



```
int fact_do(int n)
{
    int result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

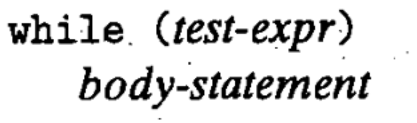
a）C 代码



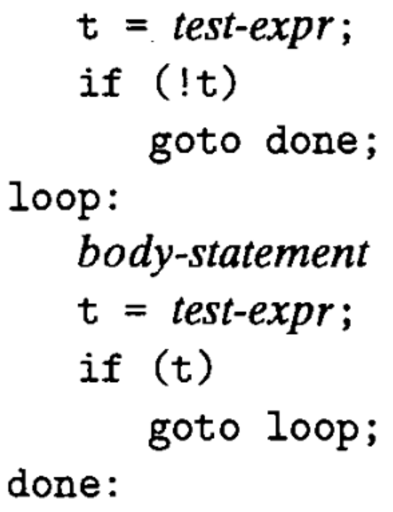
```
   Argument: n at %ebp+8
   Registers: n in %edx, result in %eax
     movl     8(%ebp), %edx      Get n
     movl     $1, %eax           Set result = 1
   .L2:                        loop:
     imull    %edx, %eax         Compute result *= n
     subl     $1, %edx           Decrement n
     cmpl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

c）对应的汇编代码

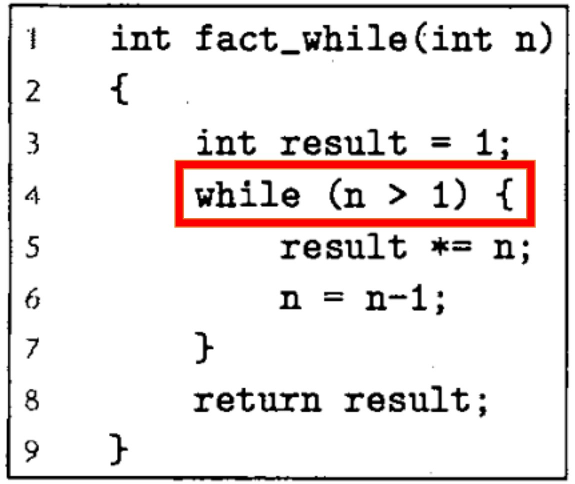
循环：while



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

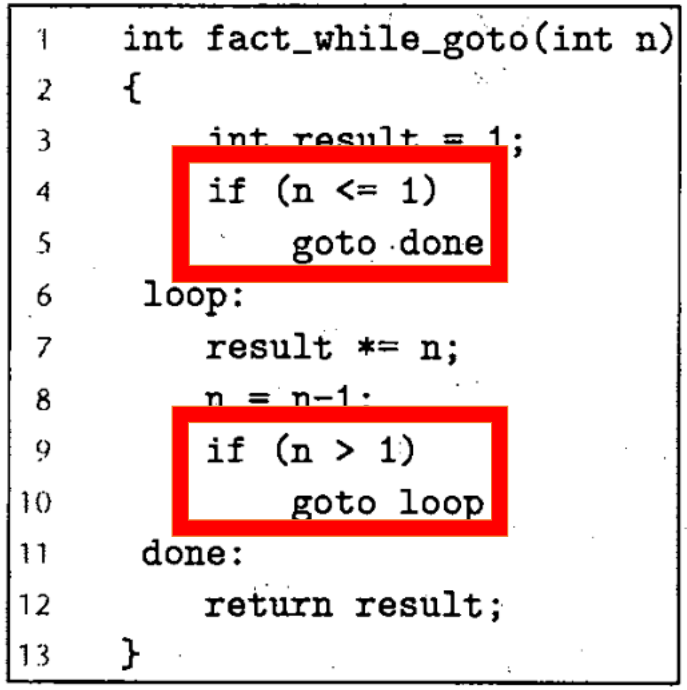


```
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
done:
```



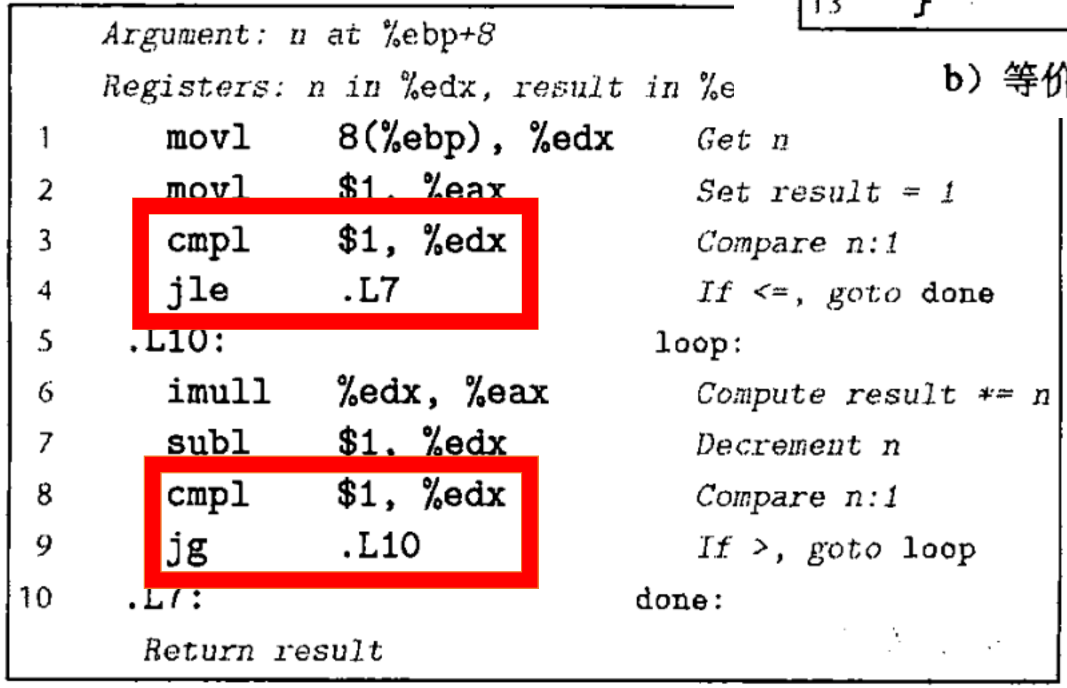
```
int fact_while(int n)
{
    int result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

a) C 代码



```
int fact_while_goto(int n)
{
    int result = 1;
    if (n <= 1)
        goto done;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
 done:
    return result;
}
```

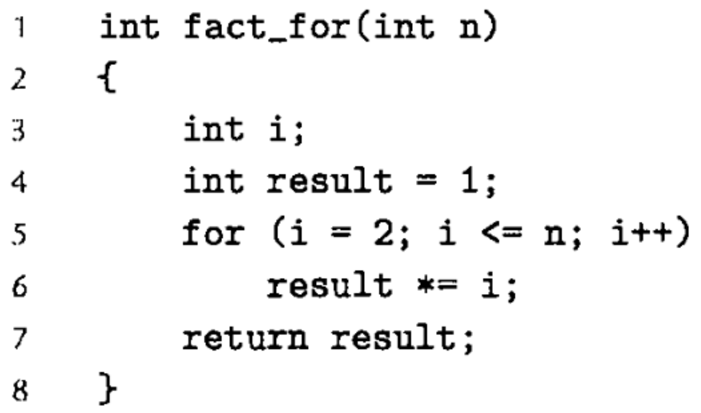
b) 等价的 goto 版本



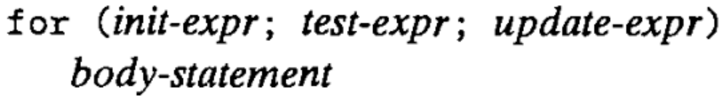
```
    Argument: n at %ebp+8
    Registers: n in %edx, result in %e
     movl    8(%ebp), %edx      Get n
     movl    $1, %eax           Set result = 1
     cmpl    $1, %edx           Compare n:1
     jle     .L7                If <=, goto done
   .L10:                      loop:
     imull   %edx, %eax         Compute result *= n
     subl    $1, %edx           Decrement n
     cmpl    $1, %edx           Compare n:1
     jg      .L10               If >, goto loop
   .L7:                       done:
    Return result
```

c) 对应的汇编代码

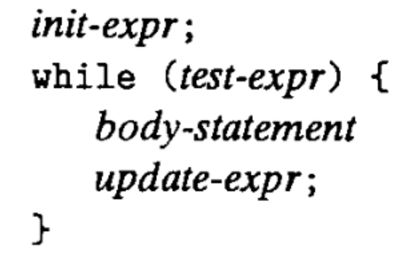
循环：for



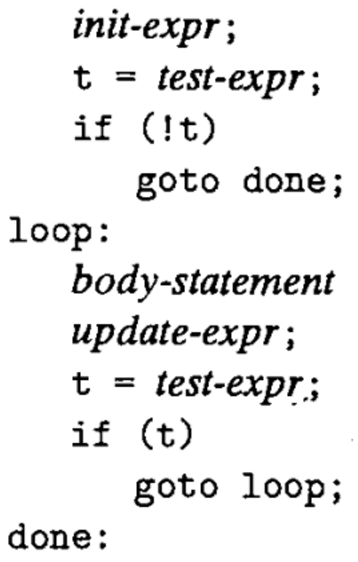
```
int fact_for(int n)
{
    int i;
    int result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```



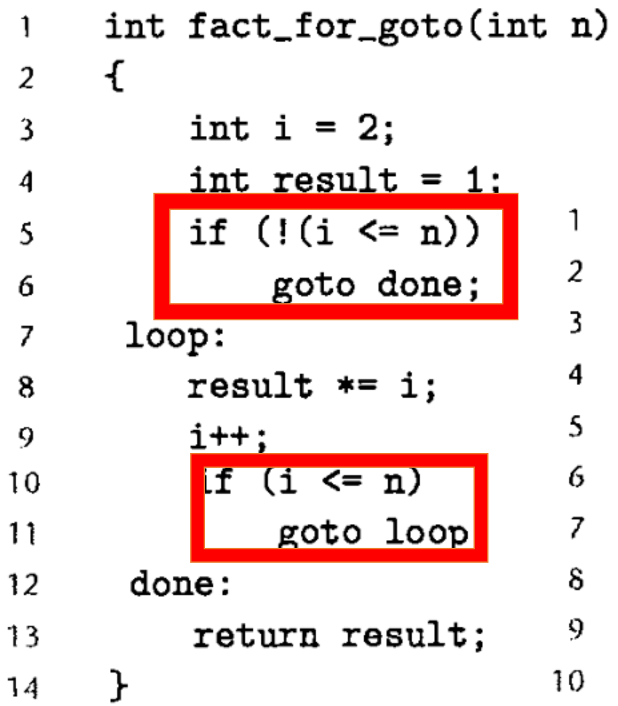
```
for (init-expr; test-expr; update-expr)
    body-statement
```



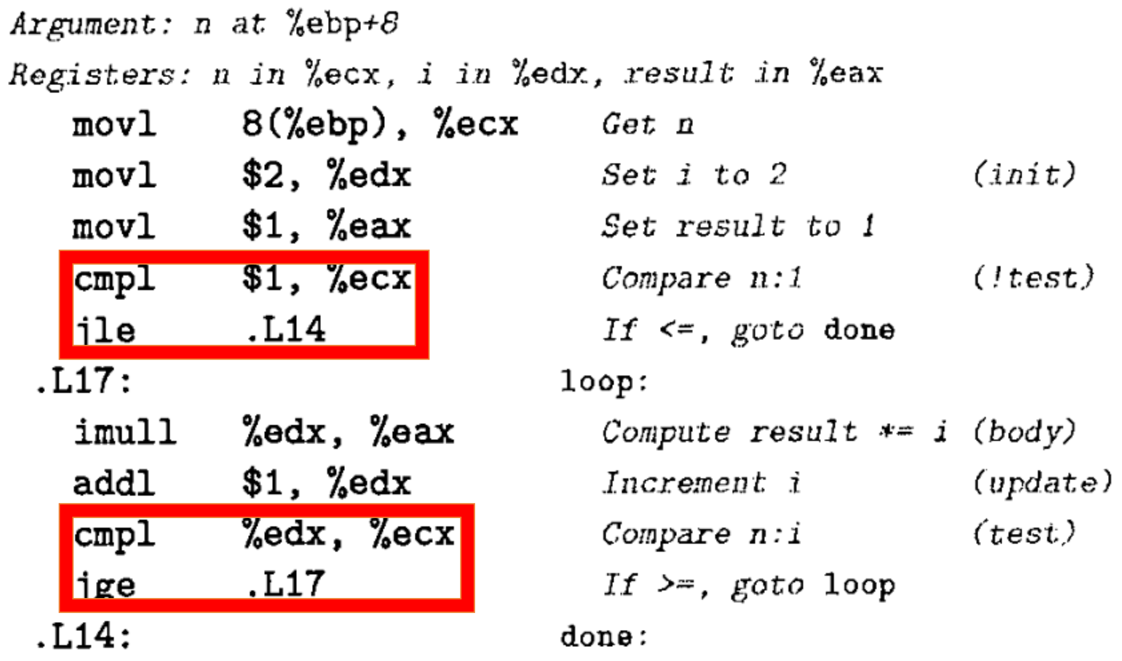
```
init-expr;
while (test-expr) {
    body-statement
    update-expr;
}
```



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



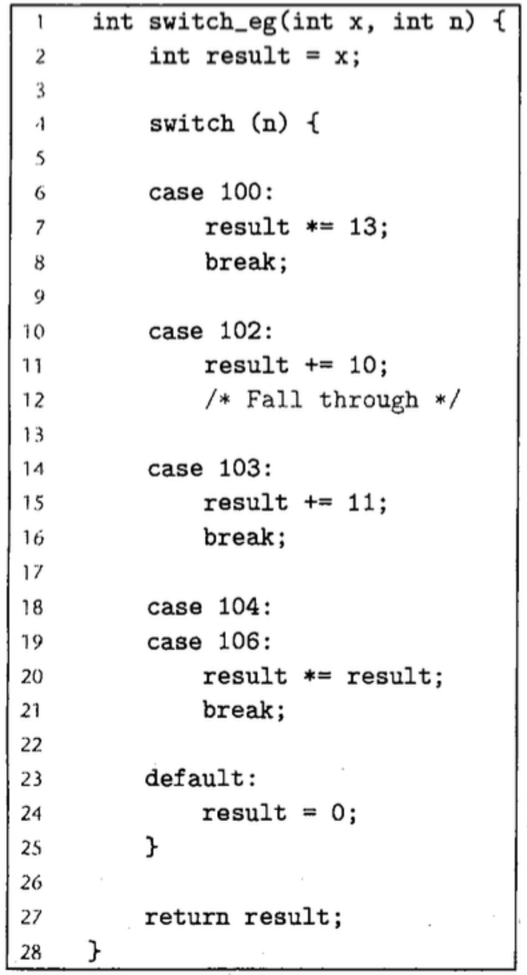
```
int fact_for_goto(int n)
{
    int i = 2;
    int result = 1;
    if (!(i <= n))
        goto done;
 loop:
    result *= i;
    i++;
    if (i <= n)
        goto loop;
 done:
    return result;
}
```



```
Argument: n at %ebp+8
Registers: n in %ecx, i in %edx, result in %eax
   movl    8(%ebp), %ecx     Get n
   movl    $2, %edx          Set i to 2              (init)
   movl    $1, %eax          Set result to 1
   cmpl    $1, %ecx          Compare n:1             (!test)
   jle     .L14              If <=, goto done
 .L17:                     loop:
   imull   %edx, %eax        Compute result *= i     (body)
   addl    $1, %edx          Increment i             (update)
   cmpl    %edx, %ecx        Compare n:i             (test)
   jge     .L17              If >=, goto loop
 .L14:                     done:
```

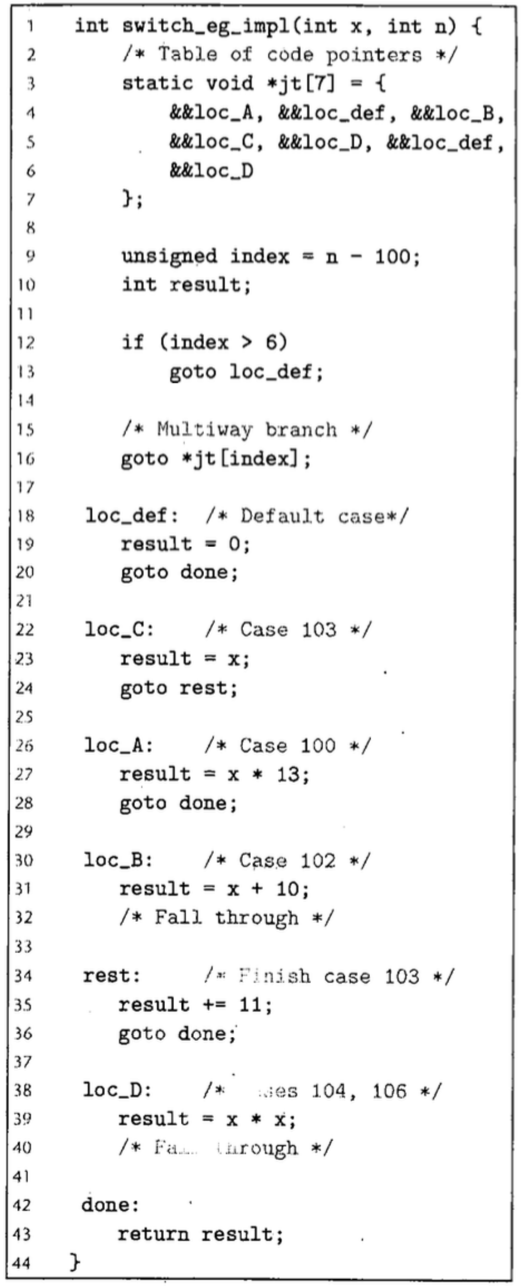
switch

- 跳转表内的数组引用



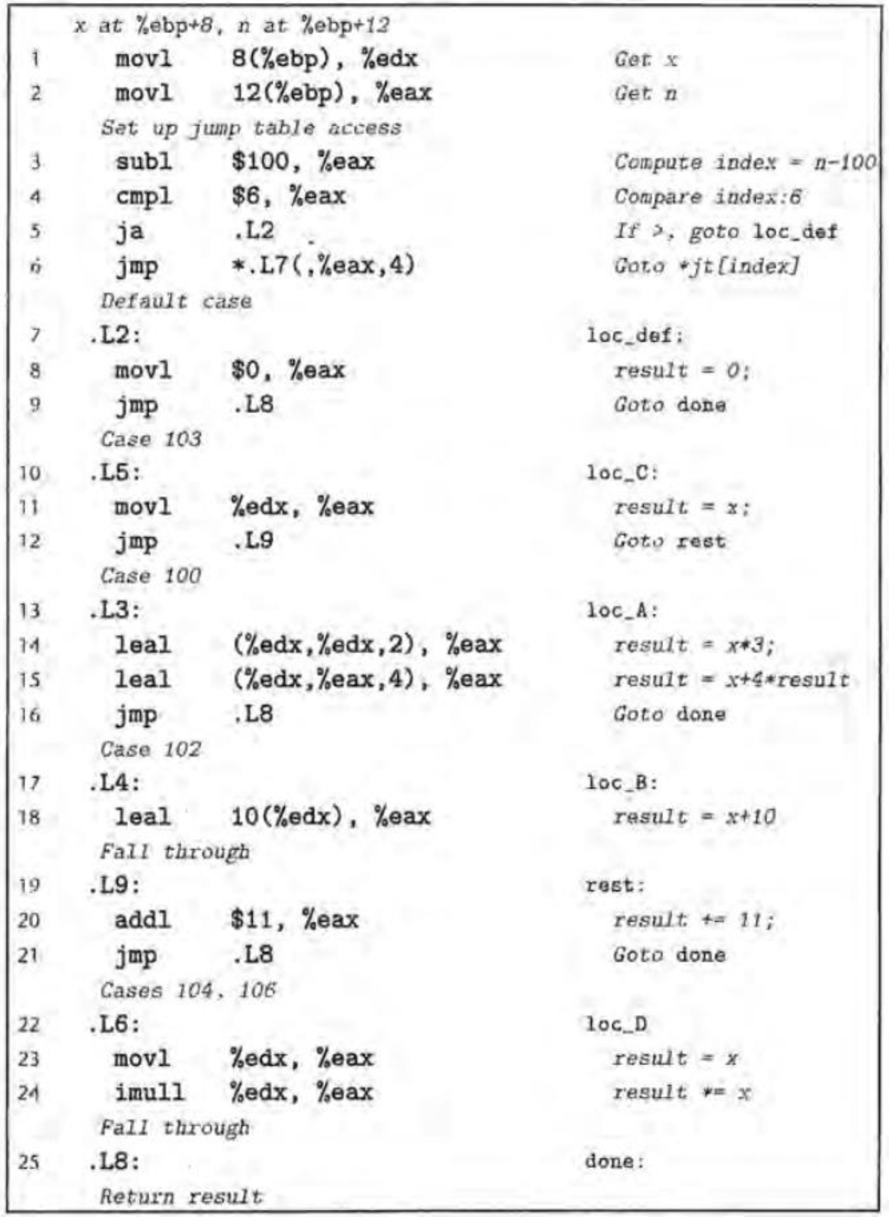
```c
int switch_eg(int x, int n) {
    int result = x;

    switch (n) {

    case 100:
        result *= 13;
        break;

    case 102:
        result += 10;
        /* Fall through */

    case 103:
        result += 11;
        break;

    case 104:
    case 106:
        result *= result;
        break;

    default:
        result = 0;
    }

    return result;
}
```

a）switch 语句



```c
int switch_eg_impl(int x, int n) {
    /* Table of code pointers */
    static void *jt[7] = {
        &&loc_A, &&loc_def, &&loc_B,
        &&loc_C, &&loc_D, &&loc_def,
        &&loc_D
    };

    unsigned index = n - 100;
    int result;

    if (index > 6)
        goto loc_def;

    /* Multiway branch */
    goto *jt[index];

 loc_def:  /* Default case*/
    result = 0;
    goto done;

 loc_C:     /* Case 103 */
    result = x;
    goto rest;

 loc_A:     /* Case 100 */
    result = x * 13;
    goto done;

 loc_B:     /* Case 102 */
    result = x + 10;
    /* Fall through */

 rest:      /* Finish case 103 */
    result += 11;
    goto done;

 loc_D:     /* Cases 104, 106 */
    result = x * x;
    /* Fall through */

 done:
    return result;
}
```

b）翻译到扩展的 C 语言

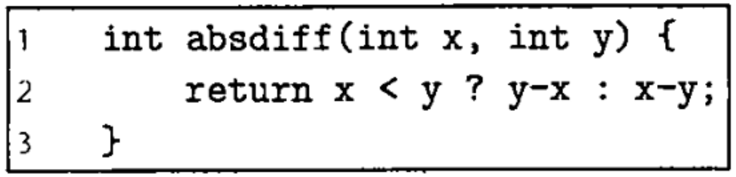


```
    x at %ebp+8, n at %ebp+12
  movl    8(%ebp), %edx            Get x
  movl    12(%ebp), %eax           Get n
 Set up jump table access
  subl    $100, %eax               Compute index = n-100
  cmpl    $6, %eax                 Compare index:6
  ja      .L2                      If >, goto loc_def
  jmp     *.L7(,%eax,4)            Goto *jt[index]
 Default case
.L2:                              loc_def:
  movl    $0, %eax                 result = 0;
  jmp     .L8                      Goto done
 Case 103
.L5:                              loc_C:
  movl    %edx, %eax               result = x;
  jmp     .L9                      Goto rest
 Case 100
.L3:                              loc_A:
  leal    (%edx,%edx,2), %eax      result = x*3;
  leal    (%edx,%eax,4), %eax      result = x+4*result
  jmp     .L8                      Goto done
 Case 102
.L4:                              loc_B:
  leal    10(%edx), %eax           result = x+10
 Fall through
.L9:                              rest:
  addl    $11, %eax                result += 11;
  jmp     .L8                      Goto done
 Cases 104, 106
.L6:                              loc_D
  movl    %edx, %eax               result = x
  imull   %edx, %eax               result *= x
 Fall through
.L8:                              done:
 Return result
```

图 3-19　图 3-18 中 switch 语句示例的汇编代码

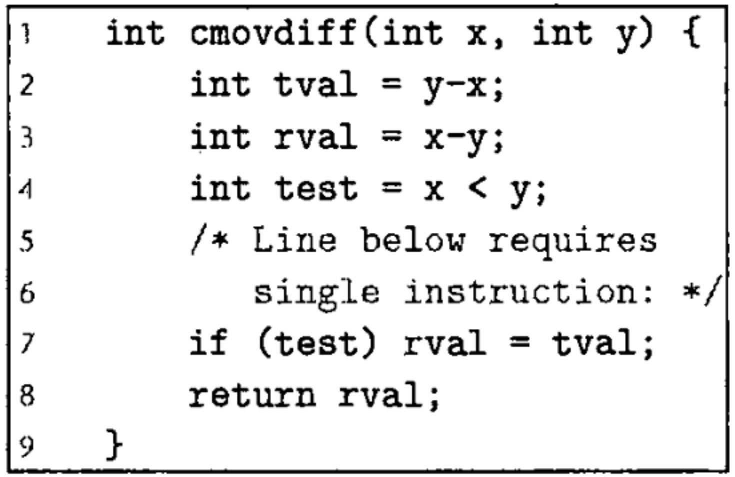
条件传送

- 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



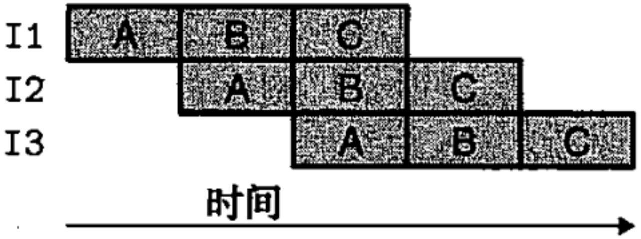
```
int absdiff(int x, int y) {
    return x < y ? y-x : x-y;
}
```

a）原始的 C 语言代码



```
int cmovdiff(int x, int y) {
    int tval = y-x;
    int rval = x-y;
    int test = x < y;
    /* Line below requires
       single instruction: */
    if (test) rval = tval;
    return rval;
}
```

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

当test-expr p为False，*p间接引用空指针

- Both values get computed
- May have undesirable effects

Computations with side effects

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

Illegal

两个分支同时改变全局变量

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

Practice

王善上

The End