计算机系统基础

程序的机器级表示(3)

王晶

jwang@ruc.edu.cn，信息楼124

2024年10月

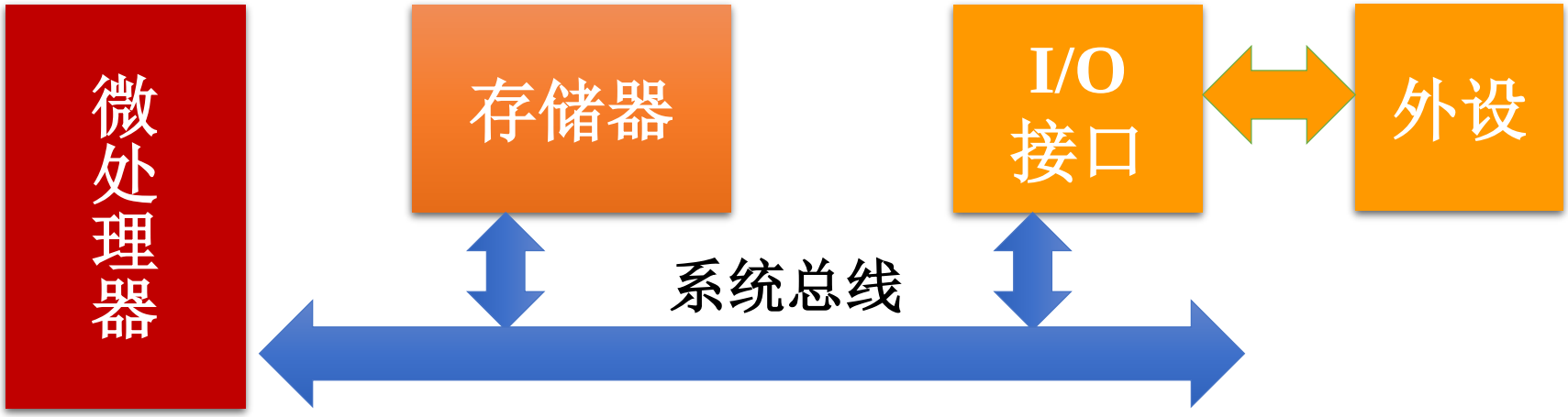
8086指令的主要类别

运算类指令

例如：加、减、乘、除，与、或、非等

传送类指令

例如：从存储器到通用寄存器，从通用寄存器到I/O接口等



控制类指令

例如：暂停处理器、清除标志位等

转移类指令

例如：无条件转移、条件转移、过程调用等

8086指令的运行结果

改变通用寄存器的内容
*如 ADD AX, DX

改变存储器单元的内容
*如 MOV [10H], CX

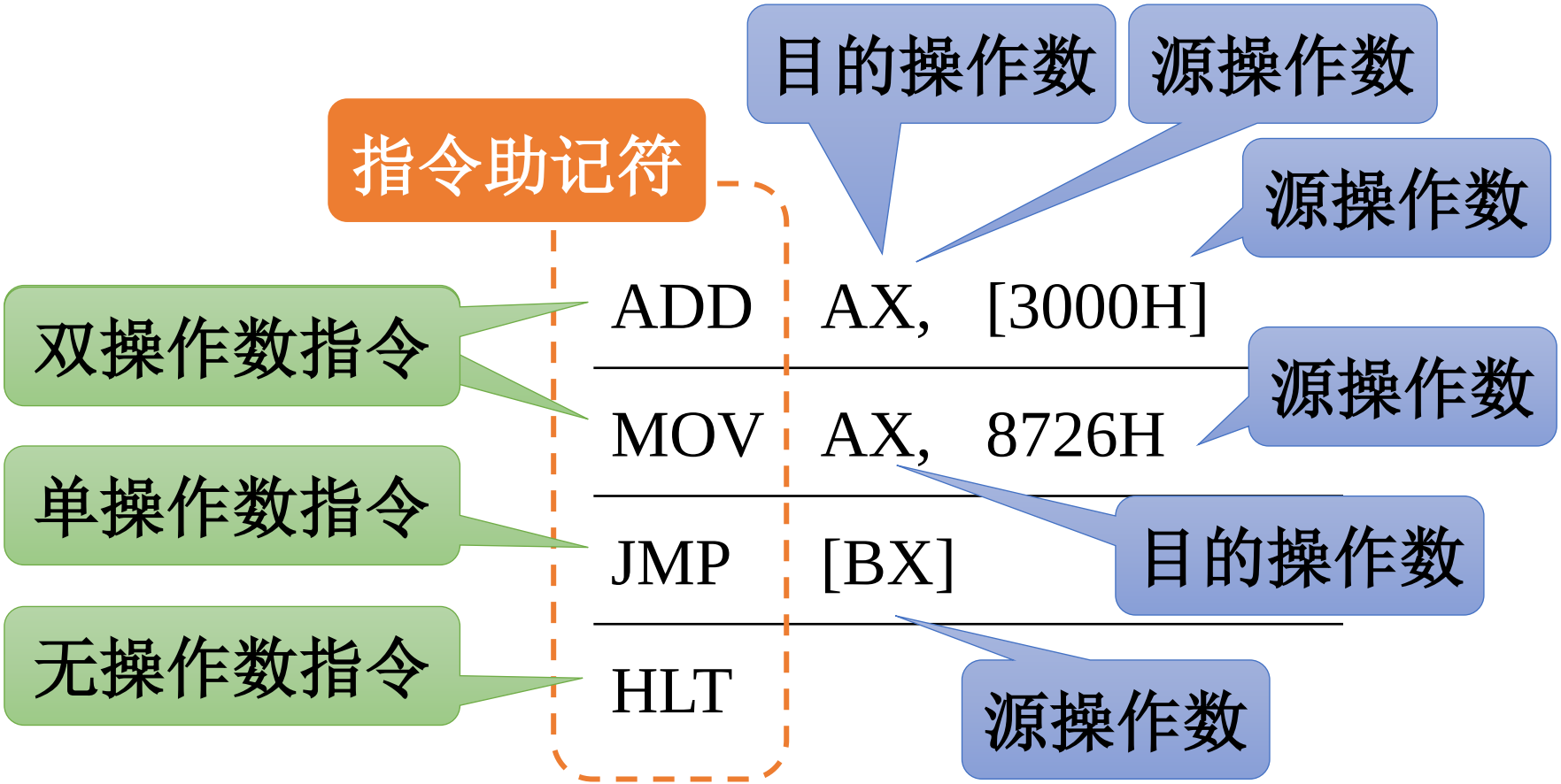
其它
……

改变标志位
*如产生进位

改变指令指针
*如 JMP [BX]

改变外设端口的内容
*如访问显示端口

8086指令的汇编形式



学习指令的注意事项

- 指令的功能——该指令能够实现何种操作。通常指令助记符就是指令功能的英文单词或其缩写形式
- 指令支持的寻址方式——该指令中的操作数可以采用何种寻址方式
- 指令对标志的影响——该指令执行后是否对各个标志位有影响，以及如何影响
- 其他方面——该指令其他需要特别注意的地方，如指令执行时的约定设置、必须预置的参数、隐含使用的寄存器等

Data Manipulation

Arithmetic Operations

Instruction	Effect	Description
lea S, D	D ← &S	Load effective address
inc D	D ← D + 1	Increment
dec D	D ← D – 1	Decrement
neg D	D ← -D	Negate
not D	D ← ~D	Complement
add S, D	D ← D + S	Add
sub S, D	D ← D – S	Subtract
cmp S, D	D – S	Subtract
imull S, D	D ← D * S	Multiply

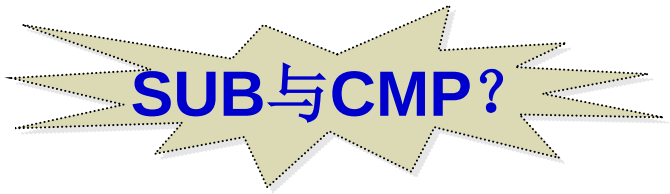
比较指令CMP

- 将目的操作数减去源操作数，差值不回送目的操作数，按照减法结果影响状态标志

  SUB reg,imm/reg/mem　　;reg－imm/reg/mem

  SUB mem,imm/reg　　;mem－imm/reg

- 根据标志状态获知两个操作数的大小关系
- 给条件转移等指令使用其形成的状态标志



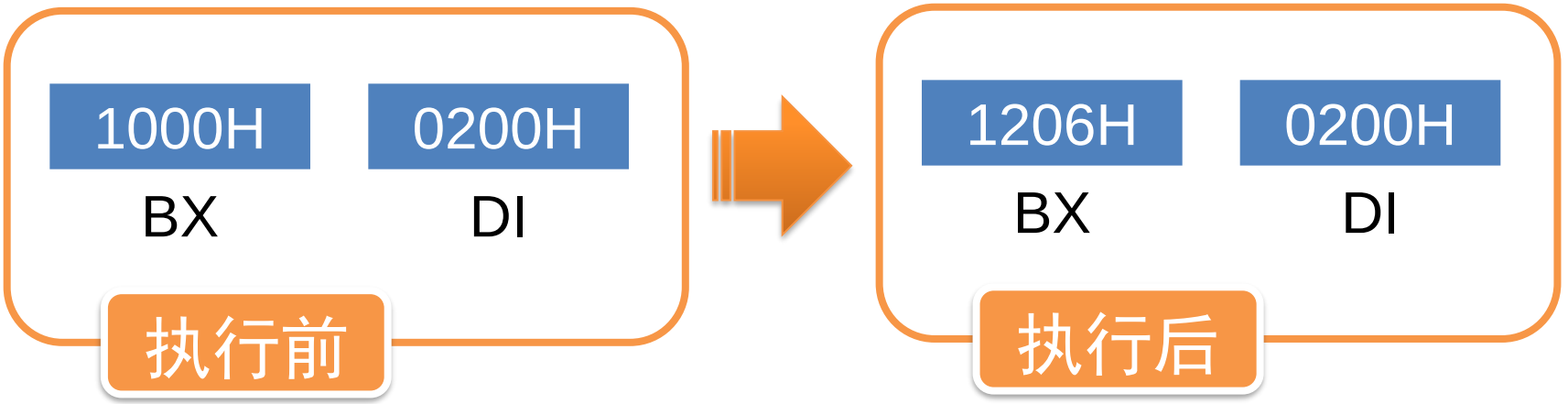
LEA指令说明

LEA指令（Load Effective Address）

- 格式：LEA　SRC , REG
- 操作：把源操作数（SRC）的有效地址（即偏移地址）装入指定寄存器（REG）
- 说明：
  - 源操作数必须是存储器操作数
  - 目的操作数必须是16位通用寄存器

LEA指令示例

LEA　(BX+DI+6H), BX



注意区别于：

MOV　(BX+DI+6H), BX

Examples for Lea Instruction

%eax holds x, %ecx holds y

Expression	Result
leal 6(%eax), %edx	6+x
leal (%eax, %ecx), %edx	x+y
leal (%eax, %ecx, 4), %edx	x+4*y
leal 7(%eax, %eax, 8), %edx	7+9*x
leal 0xA(, %ecx, 4), %edx	10+4*y
leal 9(%eax, %ecx, 2), %edx	9+x+2*y

为什么使用LEA?

- CPU 设计人员的预期用途：计算指向对象的指针
  - 也许是一个数组元素
  - 例如，要将一个数组元素仅传递给另一个函数

Assembly	C equivalent
lea (%rbx,%rdi,8), %rax	rax = &rbx[rdi]

- 编译器作者喜欢将其用于普通算术
  - 它可以在一条指令中执行复杂的计算
  - 它是 x86 仅有的三操作数指令之一
  - 它不会触及条件代码

Assembly	C equivalent
lea (%rbx,%rbx,2), %rax	rax = rbx * 3

Arithmetic and Logical Operations

Logical Operations

Instruction	Effect	Description
xor S, D	D ← D ^ S	Exclusive-or
or S, D	D ← D | S	Or
and S, D	D ← D & S	And
test S, D	D & S	And
sal k, D	D ← D << k	Left shift
shl k, D	D ← D << k	Left shift
sar k, D	D ← D >> k	Arithmetic right shift
shr k, D	D ← D >> k	Logical right shift

测试指令TEST

- 按位进行逻辑与运算，不返回逻辑与结果

  TEST reg,imm/reg/mem　　;reg ∧ imm/reg/mem

  TEST mem,imm/reg　　;mem ∧ imm/reg

- TEST指令像AND指令一样来设置状态标志
- TEST指令常用于检测一些条件是否满足，一般后跟条件转移指令，目的是利用测试条件转向不同的分支

Special Arithmetic Operations

imull S	R[%edx]:R[%eax] ←S*R[%eax]	Signed full multiply
mull S	R[%edx]:R[%eax] ←S*R[%eax]	Unsigned full multiply
Cltd	R[%edx]:R[%eax] ← SignExtend(R[%eax])	Convert to quad word
idiv S	R[%edx] ← R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] ÷ S	Signed divide
divl S	R[%edx] ←R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] ÷ S	Unsigned divide

Initially x at %ebp+8, y at %ebp+12

```
1 movl 8(%ebp), %eax
2 imull 12(%ebp)
3 pushl %edx
4 pushl %eax
```

```
1 movl 8(%ebp), %eax
2 cltd
3 idivl 12(%ebp)
4 pushl %eax
5 pushl %edx
```

算数逻辑运算

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination	Value
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull $16, (%eax, %edx, 4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx, %eax	%eax	0xFD

指令后缀

- 大多数 x86 指令可以编写，也可以不带后缀
  - `imul %rcx, %rax`
  - `imulq %rcx, %rax`

There's no difference!

- 后缀指明了操作数的大小
  - b=byte, w=short, l=int, q=long
  - If present, must match register names
- 汇编器生成的代码通常有后缀
  - (gcc –S)命令
- 反汇编生成的代码通常没有后缀
  - (objdump –d gdb 'disas')
- Intel 的手册总是省略后缀

Assembly Code for Arithmetic Expressions

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z*48;
  int t3 = t1&0xFFFF;
  int t4 = t2*t3;
  return t4;
}
```

```
movl 12(%ebp),%eax        Get y
movl 16(%ebp),%edx        Get z
addl 8(%ebp),%eax         Compute t1=x+y
leal (%edx,%edx,2),%edx   Compute 3*z
sall $4,%edx              Compute t2=48*z
andl $0xFFFF,%eax         Compute t3=t1&FFFF
imull %edx,%eax           Set t4 as return val
```

填空题 2分

设置

```
long shift_left4_rightn(long x, long n) {
    x <<= 4;
    x >>= n;
    return x;
}
```

对应的汇编为：

```
# x in %rdi, n in %rsi
Shift_left4_rightn:
    Movq      %rdi, %rax
    _____ [填空1] _____        # x <<= 4
    Movl      %esi, %ecx
    _____ [填空2] _____        # x >>= n
```

请参考注释，补全代码

作答

练习答案

- long shift_left4_rightn(long x, long n) {
  - x <<= 4;
  - x >>= n;
  - return x;
- }对应的汇编为：
- # x in %rdi, n in %rsi
- Shift_left4_rightn:
  - Movq %rdi, %rax
  - __ Salq $4, %rax_____ # x <<= 4
  - Movl %esi, %ecx
  - __ Sarq %cl, %rax_____ # x >>= n
  - 请参考注释，补全代码

填空题 4分

设置

```
Long arith2(long x, long y, long z)
{
Long t1 = [填空1]
Long t2 = [填空2]
Long t3 = [填空3]
Long t4 = [填空4]
Return t4;
}
```

- arith2:
  - #x@%rdi, y@rsi, z@rdx
  - Orq %rsi, %rdi
  - Sarq $3, %rdi
  - Notq %rdi
  - Movq %rdx, %rax
  - Subq %rdi, %rax
  - Ret

作答

课堂练习

```
Long arith2(long x, long y, long z) {
    Long t1 = x | y;
    Long t2 = t1 >> 3;
    Long t3 = ~t2;
    Long t4 = z-t3;
    Return t4;
}
```

- Arith2:
  - #x@%rdi, y@rsi, z@rdx
  - orq %rsi, %rdi
  - sarq $3, %rdi
  - notq %rdi
  - movq %rdx, %rax
  - subq %rdi, %rax
  - ret
- 填写空缺的C语言语句

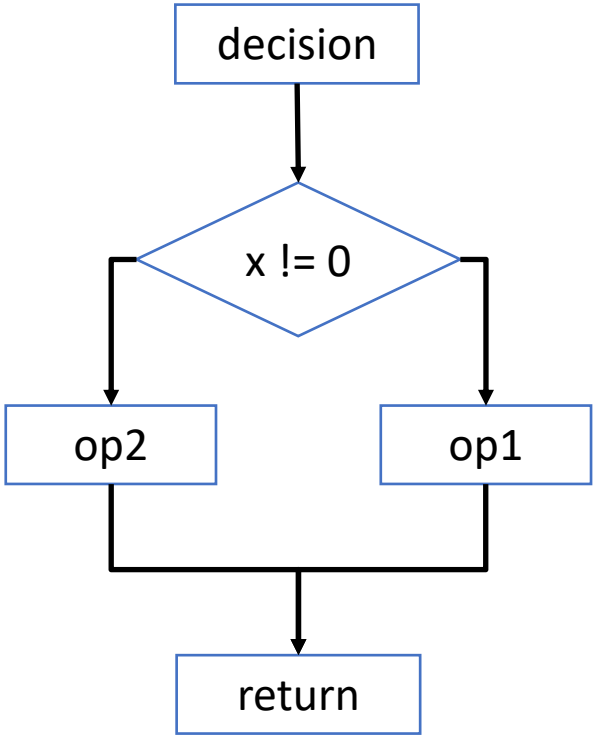
控制的类型

- 控制：条件代码
- 条件分支
- 循环
- switch 语句

Control flow

```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```



Control flow in assembly language

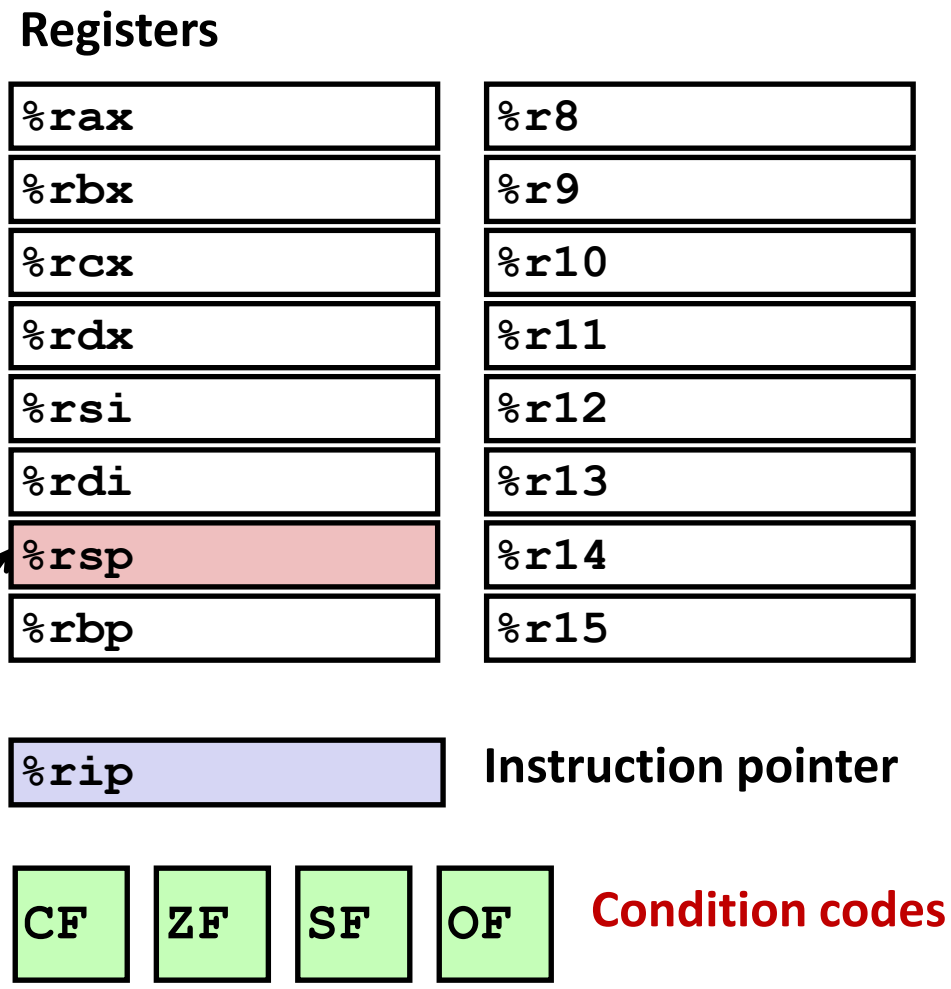
```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```

```
decision:
        subq    $8, %rsp
        testl   %edi, %edi
        je      .L2
        call    op1
        jmp     .L1
.L2:
        call    op2
.L1:
        addq    $8, %rsp
        ret
```

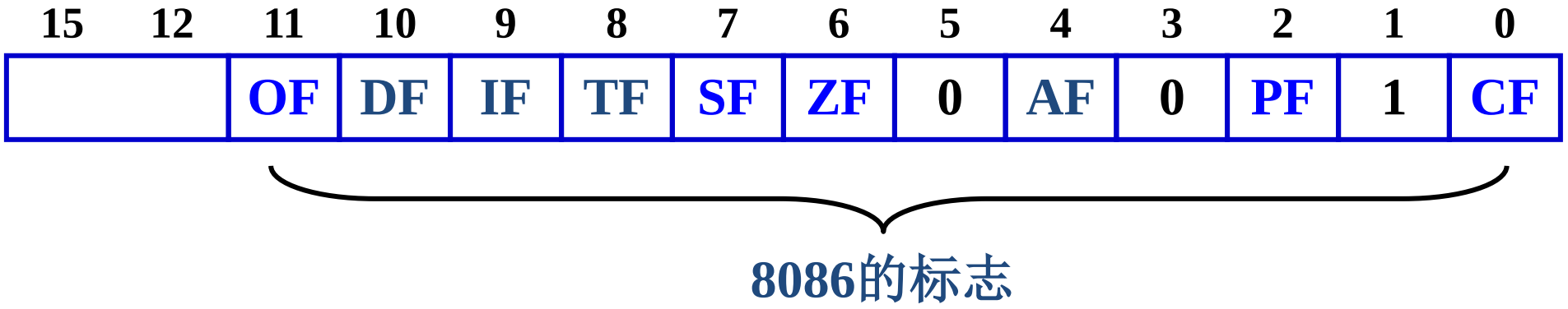
Processor State (x86-64, Partial)

- **Information about currently executing program**
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)



状态标志

- 状态标志是处理器最基本的标志
- 一方面：作为加减运算和逻辑运算的辅助结果
- 另一方面：构成各种条件，实现程序分支



进位标志CF（Carry Flag）

- 当加减运算结果的最高有效位有进位（加法）或借位（减法）时，进位标志置1，即CF＝1；否则CF＝0
- 针对无符号整数，判断加减结果是否超出表达范围
  - N个二进制位表达无符号整数的范围：

    $0 \sim 2^N-1$

  - 8位：0～＋255
  - 16位：0～＋65535
  - 32位：0～＋2^{32}－1

CF set when

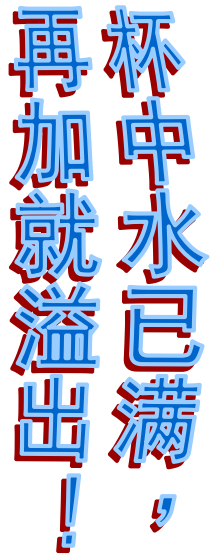
For unsigned arithmetic, this reports overflow

溢出标志OF（Overflow Flag）

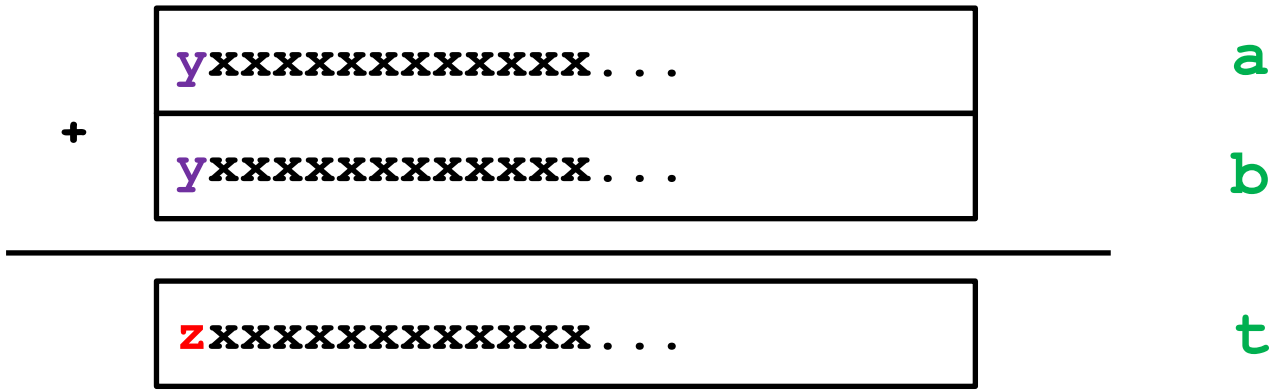
- 有符号数加减结果有溢出，则OF＝1；否则OF＝0
- 针对有符号整数，判断加减结果是否超出表达范围
  - N个二进制位（补码）表达有符号整数的范围：

    $$-2^{N-1} \sim 2^{N-1}-1$$

  - 8位：－128～＋127
  - 16位：－32768～＋32767
  - 32位：$-2^{31} \sim +2^{31}-1$



OF set when



z = ~y

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

For signed arithmetic, this reports overflow

进位和溢出的区别

- 进位标志反映无符号整数运算结果是否超出范围
  - 有进位，加上进位或借位后运算结果仍然正确
- 溢出标志反映有符号整数运算结果是否超出范围
  - 有溢出，运算结果已经不正确
- 处理器按照无符号整数求得结果
  - 设置进位标志CF
  - 设置溢出标志OF
- 程序员决定
  - 操作数是无符号数，关心进位
  - 操作数是有符号数，注意溢出

零标志ZF（Zero Flag）

- 运算结果为0，则ZF＝1，否则ZF＝0

结果是0，
ZF标志不是0！

举例

- 8位二进制数相加：

  00111010＋01111100＝10110110

  结果不是0，ZF＝0

- 8位二进制数相加：

  10000100＋01111100＝[1]00000000

  结果是0，ZF＝1

进位　结果

符号标志SF（Sign Flag）

- 运算结果最高位为1，则SF＝1；否则SF＝0

最高位＝符号位＝SF

举例

- 8位二进制数相加：

  00111010＋01111100＝10110110

  最高位＝1：SF＝1

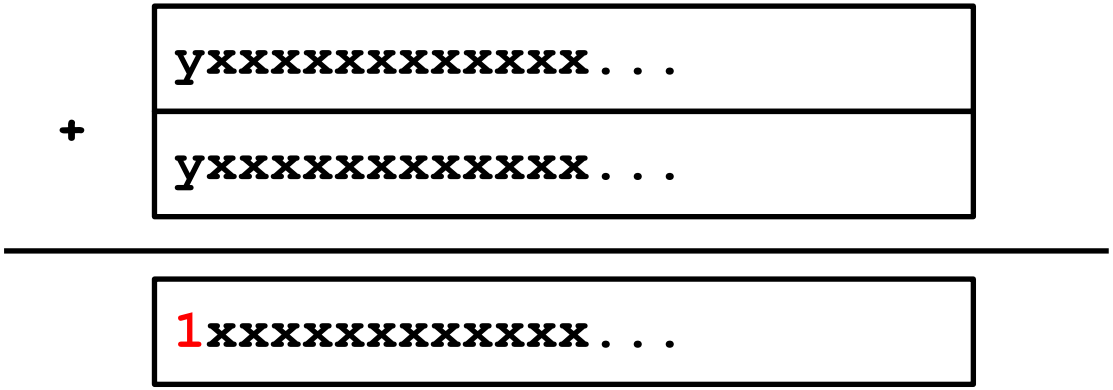
- 8位二进制数相加：

  10000100＋01111100＝[1]00000000

  最高位＝0：SF＝0

进位　结果

SF set when



For signed arithmetic, this reports when result is a negative number

设置标志位

- 算数运算指令隐式设置

  `addl` *Src,Dest*

  C analog: `t = a+b`

  - CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  - ZF set if `t == 0`
  - SF set if `t < 0`
  - OF set if two's complement overflow

    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

设置标志位

- 比较指令显示设置
  - **`cmpl`** ***Src2,Src1***
  - **`cmpl b,a`** like computing `a-b` without setting destination
  - CF set if carry out from most significant bit
    - Used for unsigned comparisons
  - ZF set if `a == b`
  - SF set if `(a-b) < 0`
  - OF set if two's complement overflow

```
(a>0 && b<0 && (a-b)<0) ||
(a<0 && b>0 && (a-b)>0)
```

设置标志位

- Test指令显式设置
  - **`testl`** ***Src2,Src1***
  - Sets condition codes based on value of *Src1* `&` *Src2*
    - Useful to have one of the operands be a mask
  - **`testl b,a`** like computing `a&b` without setting destination
  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

标志位的特殊规则

- lea instruction (数据移动类的都不影响EFlags)
  - has no effect on condition codes
- Xorl instruction
  - The carry CF and overflow OF flags are set to 0
- Shift instruction
  - Carry flag is set to the last bit shifted out
  - Overflow flag is set to 0
- Inc/dec instruction
  - Do NOT set carry flag
- Mull/imull instruction
  - CF=OF=0 iff the first half of result is useless; else, CF=OF=1

例：加法运算

判断SF, ZF, CF, OF

```
movw $0075h, %ax                ;ax=0075h
movw $01a2h, %cx                ;cx=01a2h
addw, %cx, %ax                  ;ax=0217h
```

SF=0、ZF=0、CF=0 、 OF=0

```
movw $77ach, %ax,               ;ax=77ach
movw $4b35h, %cx                ;cx=4b35h
addw %cx, %ax                   ;ax=c2e1h
```

SF=1、ZF=0、CF=0 、 OF=1

例：减法运算

```
movl $75h, %eax                ;ax=00000075h
movl $1a2h, %ecx                ;cx=000001a1h
subl %ecx, %eax                ;ax=fffffed3h
```

SF=1、ZF=0、CF=1(借位) 、 OF=0

```
movw $c36eh, %ax                ;ax=c36eh
movw $ef00h, (%ebx)             ; (%ebx)=ef00h
subw %ax, (%ebx)              ; (%ebx)=2b92h
```

SF=0、ZF=0、CF=0 、 OF=0

读取标志位

- 标志位不能直接读写（像其他寄存器一样用move等指令）
- 写入：主要是通过相关的运算来设置标识位
- 读取：
  - 通过set系列指令，根据eflags状态来设置参数的值
    - Sete %al
  - 通过一些特定指令来读取判断（条件跳转语句，条件传送数据等）
- 特殊方法：pushf, popf可以用于修改eflags

读取标志位

Instruction	Synonym	Effect	Set Condition
Sete	Setz	ZF	Equal/zero
Setne	Setnz	~ZF	Not equal/not zero
Sets		SF	Negative
Setns		~SF	Nonnegative
Setl	Setnge	SF^OF	Less
Setle	Setng	(SF^OF)|ZF	Less or Equal
Setg	Setnle	~(SF^OF)&~ZF	Greater
Setge	Setnl	~(SF^OF)	Greater or Equal
Seta	Setnbe	~CF&~ZF	Above
Setae	Setnb	~CF	Above or equal
Setb	Setnae	CF	Below
Setbe	Setna	CF|ZF	Below or equal

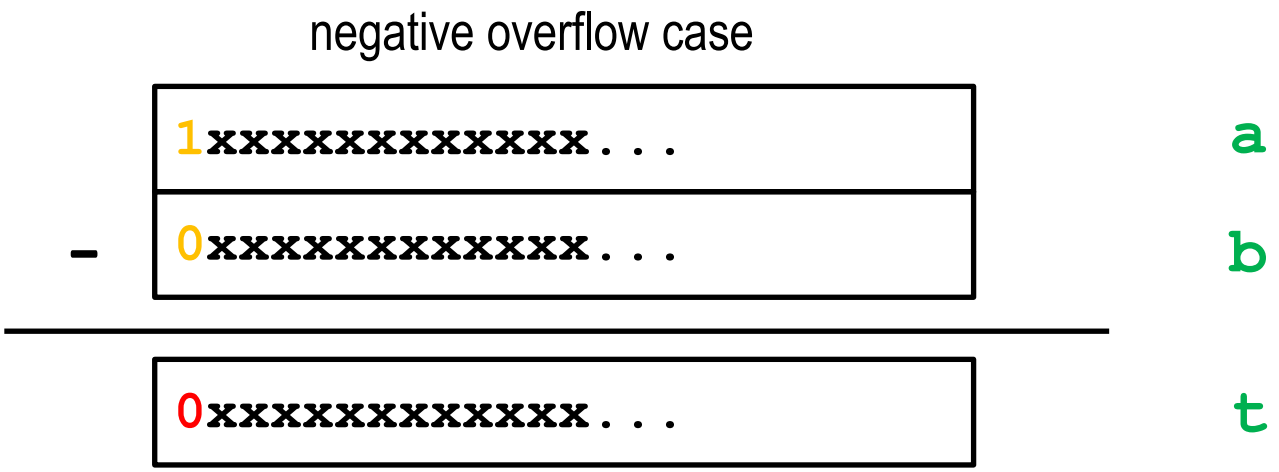
有符号数

无符号数

Example: setl (Signed <)

- Condition: SF^OF

SF	OF	SF ^ OF	Implication
0	0	0	No overflow, so SF implies not <
1	0	1	No overflow, so SF implies <
0	1	1	Overflow, so SF implies negative overflow, i.e. <
1	1	0	Overflow, so SF implies positive overflow, i.e. not <



标志位用于比较

例：

				flags				interpretation	
	operand1	operand2	difference	CF	OF	SF	ZF	signed	unsigned
1	3B	3B	00	0	0	0	1	*op1=op2*	*op1=op2*
2	3B	15	26	0	0	0	0	*op1>op2*	*op1>op2*
3	15	3B	DA	1	0	1	0	*op1<op2*	op1<op2
4	F9	F6	03	0	0	0	0	*op1>op2*	*op1>op2*
5	F6	F9	FD	1	0	1	0	*op1<op2*	op1<op2
6	15	F6	1F	1	0	0	0	*op1>op2*	op1<op2
7	F6	15	E1	0	0	1	0	*op1<op2*	*op1>op2*
8	68	A5	C3	1	1	1	0	*op1>op2*	op1<op2
9	A5	68	3D	0	1	0	0	*op1<op2*	*op1>op2*
								of xor sf	cf

读取标志位(Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

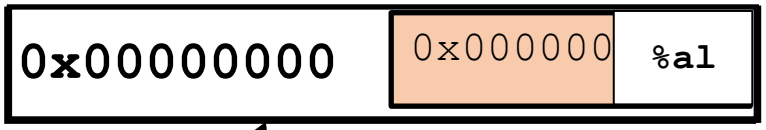
Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

显式读取标志位(Cont.)

Beware weirdness **movzbl** (and others)

movzbl %al, %eax



Zapped to all 0's

Use(s)
Argument **x**
Argument **y**
Return value

```
    cmpq    %rsi, %rdi    # Compare x:y
    setg    %al           # Set when >
    movzbl  %al, %eax     # Zero rest of %rax
    ret
```

Exercise

`cmpq b,a` like computing `a-b` w/o setting dest

- **CF set** if carry/borrow out from most significant bit (used for unsigned comparisons)
- **ZF set** if `a == b`
- **SF set** if `(a-b) < 0` (as signed)
- **OF set** if two's-complement (signed) overflow

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (signed)
`setge`	`~(SF^OF)`	Greater or Equal (signed)
`setl`	`SF^OF`	Less (signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzblq %al, %eax
```

%rax	SF	CF	OF	ZF

Note: `setl` and `movzblq` do not modify condition codes

课堂练习

- int comp (data_t a, data_t b) {
  - return a COMP b;
- }
- 64位机环境下，对于下面每个汇编指令序列，确定哪种数据类型data_t和比较操作COMP会导致编译器产生如下代码：
  - A. cmpl %esi, %edi; setl %al
  - B. cmpw %si, %di; setge %al
  - C. cmpb %sil, %dil; setbe %al
  - D. cmpq %rsi, %rdi; setne %al

练习答案

- int comp (data_t a, data_t b) {
  - return a COMP b;
- }
- 64位机环境下，对于下面每个汇编指令序列，确定哪种数据类型data_t和比较操作COMP会导致编译器产生如下代码：
  - A. cmpl %esi, %edi; setl %al int, <
  - B. cmpw %si, %di; setge %al short, >=
  - C. cmpb %sil, %dil; setbe %al unsigned char, <=
  - D. cmpq %rsi, %rdi; setne %al long, unsigned long, 或指针，!=

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

Control

- 程序的执行
  - 数据流 (Accessing and operating data)
  - 控制流 (control the sequence of operations)
- 默认情况顺序执行
  - 指令按照其在程序中出现的顺序执行
- 改变控制流
  - Jump 跳转指令，产生分支执行
  - 分支语句对性能有一定影响，分支预测失败会有较大的惩罚（根源在于内存访问速度远低于CPU运算速度）

Unconditional jump

- Jumps unconditionally
- Direct jump: jmp label
  - jmp .L
- Indirect jump: jmp *Operand
  - jmp *%eax
  - jmp *(%eax)

Conditional jump

- 或者跳转，或者继续顺序执行
  - 取决于一定的condition codes
  - Jz(je), jnz(jne)
  - Js, jns
  - 有符号数：jl(jnge), jg(jnle), jle(jng), jge(jnl)
  - 无符号数：ja(jnbe), je(jnae), jae(jnb), jbe(jna)
- 都是direct jump类型

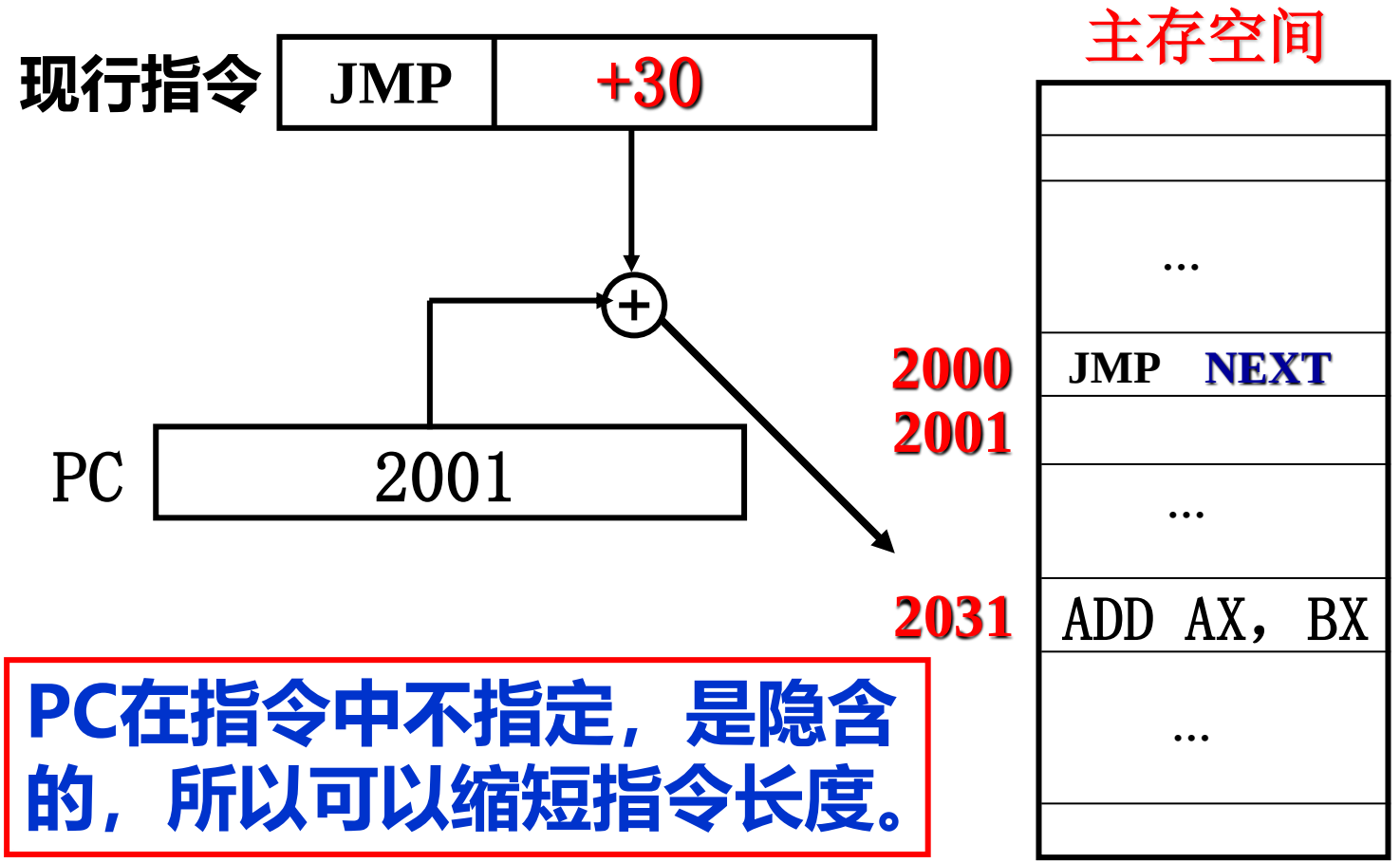
Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

相对寻址

- 指令的地址由程序计数器PC的内容和指令地址码相加得到
- 主要用于转移指令，缩短指令长度。



PC在指令中不指定，是隐含的，所以可以缩短指令长度。

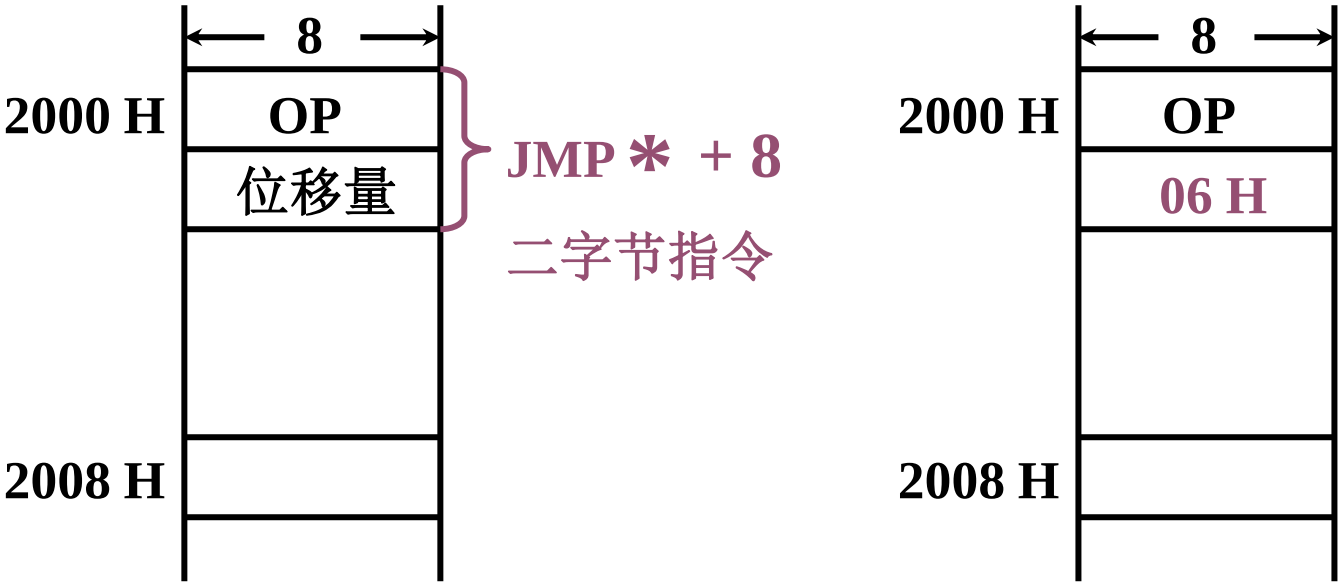
相对寻址举例

```
        LDA    # 0
        LDX    # 0
M       ADD    X, D
M+1     INX
M+2     CPX    # N
M+3     JNE    M      ——→  * – 3
        DIV    # N
        STA    ANS
```

指令 BNE *– 3 操作数的有效地址为

EA = (M+3) – 3 = M

按字节寻址的相对寻址举例



设 当前指令地址 PC = 2000H

转移后的目的地址为 2008H

因为 取出 JMP * + 8 后 PC = 2002H

转移后的目的地址为 1FFDH

第二个字节为 FBH

目标重定位的PC相对寻址

1	8:	7e 0d	jle	17<silly+0x17>
2	a:	89 d0	mov	%edx, %eax *dest1:*
3	c:	c1 f8	sar	%eax
4	e:	29 c2	sub	%eax, %edx
5	10:	8d 14 52	lea	(%edx, %edx, 2), %edx
6	13:	85 d2	test	%edx, %edx
7	15:	7f f3	jg	a<silly+0x10>
8	17:	89 d0	movl	%edx, %eax *dest2:*

d+a = 17

17+f3(-d) =a

目标重定位的PC相对寻址

1	804839:	7e 0d	jle	17<silly+0x17>
2	804839e:	89 d0	mov	%edx, %eax *dest1:*
3	80483a0:	c1 f8	sar	%eax
4	80483a2:	29 c2	sub	%eax, %edx
5	80483a4:	8d 14 52	lea	(%edx, %edx, 2), %edx
6	80483a7:	85 d2	test	%edx, %edx
7	80483a9:	7f f3	jg	a<silly+0x10>
8	80483ab:	89 d0	movl	%edx, %eax *dest2:*

d+804839e = 80483ab

80483ab+f3(-d) = 804839e

课堂练习

- 下列XXXX的值是什么？
  - 1)
    - 0x804828f:　　74 05　　je　XXXX
    - 0x8048291: e8 1e 00 00 00　　call 80482b4
  - 2)
    - 0x8048357: 72 e7　　jb XXXX
    - 0x8048359: c6 05 10 a0 04 08 01　movb $0x1, 0x804a010

练习答案

- 下列XXXX的值是什么？
  - 1)
    - 0x804828f:　　74 05　　je　XXXX (0x8048296)
    - 0x8048291: e8 1e 00 00 00　　call 80482b4
  - 2)
    - 0x8048357: 72 e7　　jb XXXX (0x8048340)
    - 0x8048359: c6 05 10 a0 04 08 01　movb $0x1, 0x804a010

课堂练习

- 3)
  - XXXX: 74 12　　je 8048391
  - XXXX: b8 00 00 00 00　　mov $0x0, %eax
- 4)
  - 80482bf: e9 e0 ff ff ff　　jmp XXXX
  - 80482c4: 90　　nop

练习答案

- 3)
  - 0x804837d: 74 12　　je 8048391
  - 0x804837f: b8 00 00 00 00　　mov $0x0, %eax
- 4)
  - 80482bf: e9 e0 ff ff ff　　jmp 0x80482a4
  - 80482c4: 90　　nop

条件跳转

- Generation

```
server> gcc -Og -S -fno-if-conversion control.c
```

Get to this shortly

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

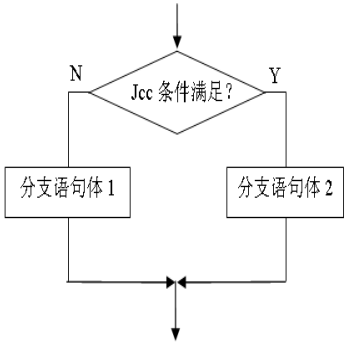
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

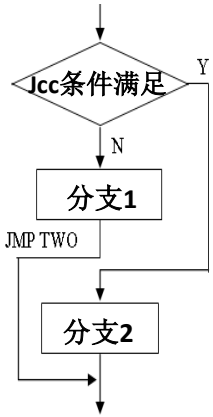
Expressing with Goto Code

- C allows **goto** statement
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```





条件表达式翻译-分支

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

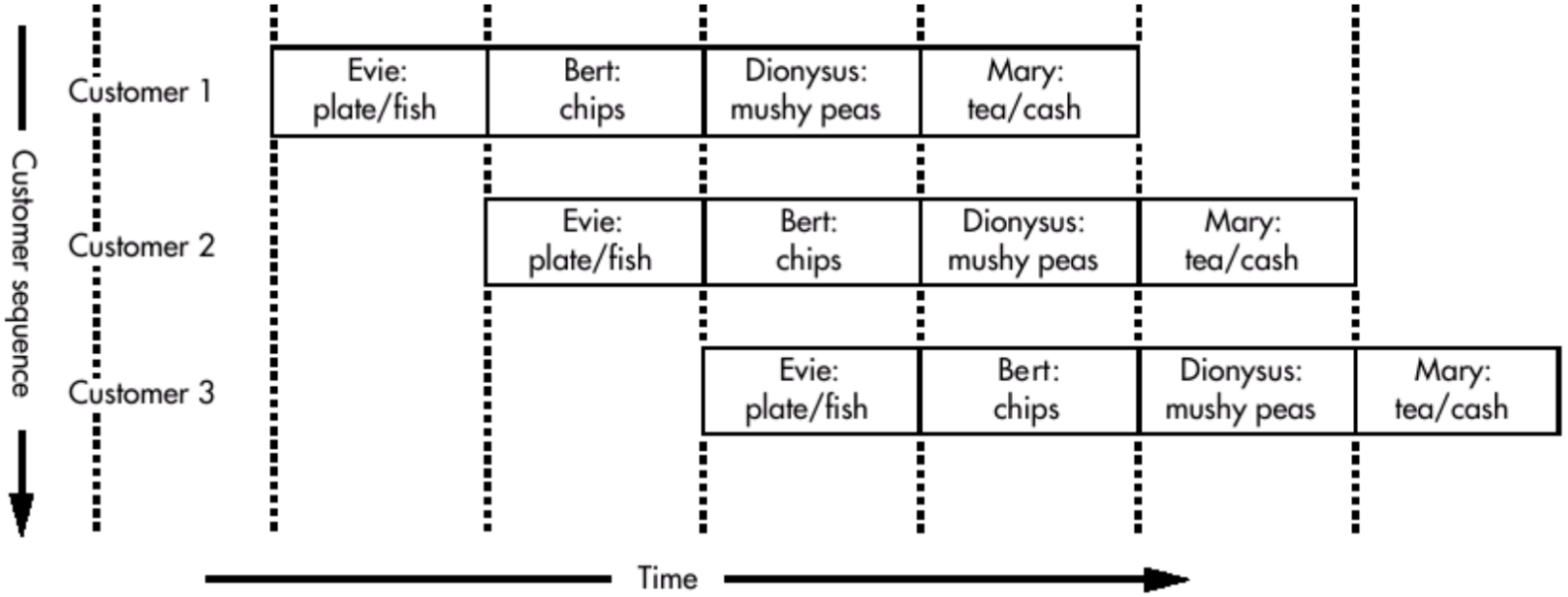
```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

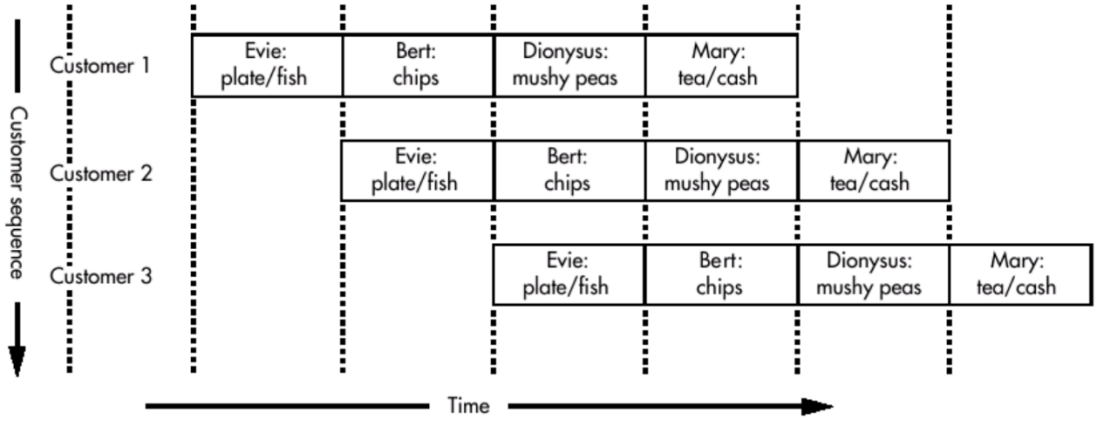
流水线（pipeline）

- Evie炸鱼店
  - 提供：炸鳕鱼、炸薯片、豌豆粥、茶
  - 排队问题
  - 加长柜台，排队移动中依次得到各种食物



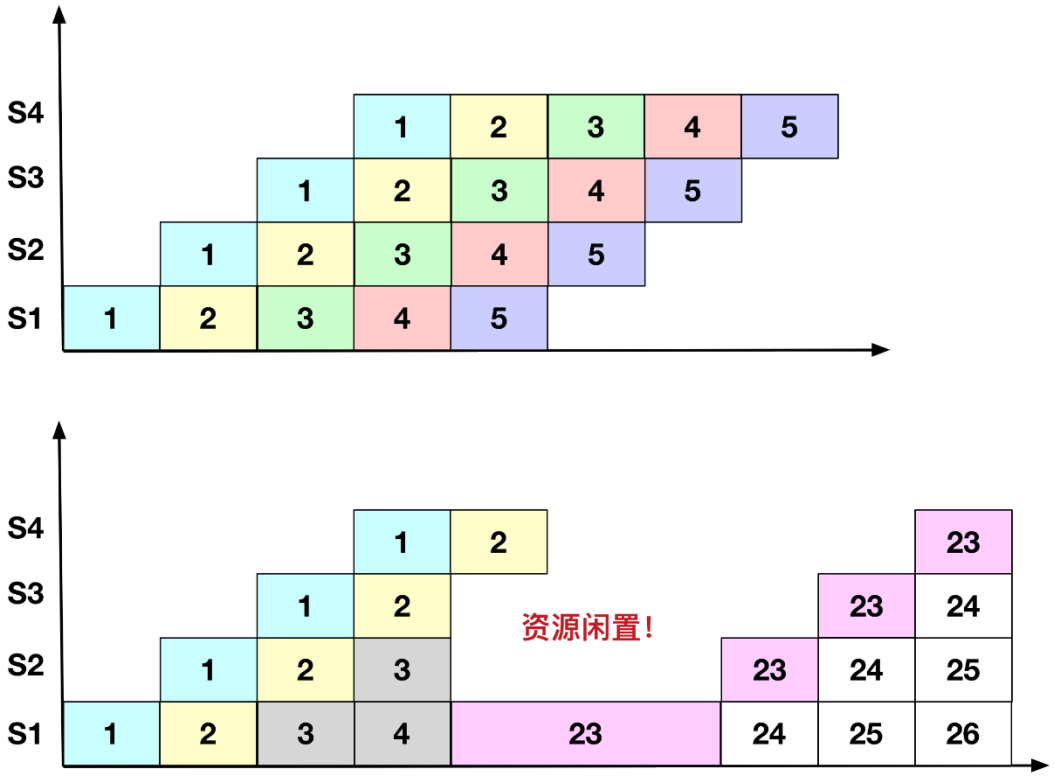
流水线（pipeline）

- 优点
  - 资源充分利用，减少排队时间
  - 每个工人的工作简单，容易培训，成本低，不容易出错
- 挑战：
  - Daphne和Lola一起来，Daphne不买茶，则Lola不买薯片。所以Lola驻足观望Daphne是否买茶（气泡）
  - 对信誉不好的Cyril，要等Mary点完钱，Evie才给他盛炸鱼（Cyril要用前面三个顾客用的资源——Mary，资源冲突）



流水线（pipeline）

- 如果分支预测失败
  - 需要再去加载新的指令，很长时间内，流水线各个环节没有指令需要运行，处于空闲状态，性能下降



Conditional Move Example

```
long absdiff
   (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax  # x
    subq    %rsi, %rax  # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx  # eval = y-x
    cmpq    %rsi, %rdi  # x:y
    cmovle  %rdx, %rax  # if <=, result = eval
    ret
```

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**

Computations with side effects

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

Illegal

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

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

“Do-While” Loop Example

Loop是标号
冒号后指令的地址

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1’s in argument x (“popcount”)
- Use conditional branch to either continue looping or to exit loop

x86 being CISC has a popcount instruction

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

- Body:

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

"Do-While" Loop Compilation

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %eax    #  result = 0
.L2:                        # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    #  t = x & 0x1
    addq    %rdx, %rax  #  result += t
    shrq    %rdi        #  x >>= 1
    jne     .L2         #  if (x) goto loop
    rep; ret
```

General "While" Translation #1

- "Jump-to-middle" translation
- Used with `-Og`

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Jump to Middle

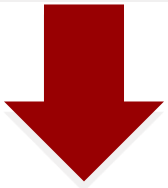
```
long pcount_goto_jtm
   (unsigned long x) {
   long result = 0;
   goto test;
 loop:
   result += x & 0x1;
   x >>= 1;
 test:
   if(x) goto loop;
   return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General "While" Translation #2

While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

- "Do-while" conversion
- Used with `-O1`

While Loop Example #2

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Do-While Version

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

- Initial conditional guards entrance to loop
- Compare to do-while version of function
  - Removes jump to middle. When is this good or bad?

“For” Loop Form

General Form

```
for (Init; Test; Update )
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update )
      Body
```

While Version

```
Init;
while (Test ) {
      Body
      Update;
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

"For" Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

- Initial test can be optimized away – why?

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

Today

- Control: Condition codes
- Conditional branches
- Loops
- **Switch Statements**

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

```
long switch_eg
    (long x, long y, long z)
{
     long w = 1;
     switch(x) {
     case 1:
          w = y*z;
          break;
     case 2:
          w = y/z;
          /* Fall Through */
     case 3:
          w += z;
          break;
     case 5:
     case 6:
          w -= z;
          break;
     default:
          w = 2;
     }
     return w;
}
```

Jump Table Structure

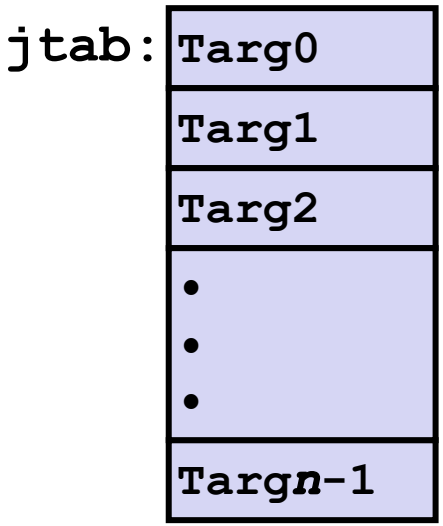
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

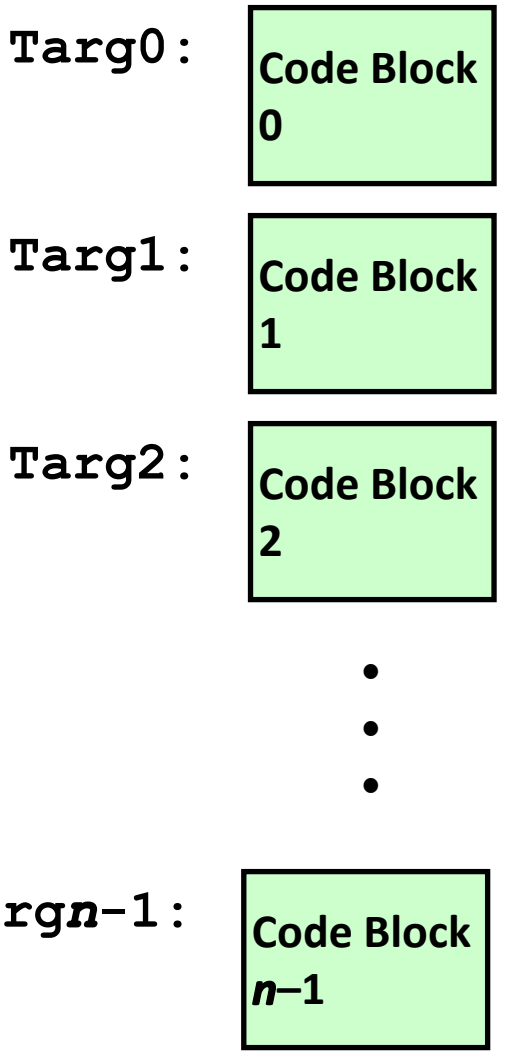
Translation (Extended C)

```
goto *JTab[x];
```

Jump Table



Jump Targets



Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

What range of values takes default?

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Note that w not initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump →

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at **`.L4`**

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

- Jumping
  - **Direct:** **`jmp .L8`**
  - Jump target is denoted by label **`.L8`**
  - **Indirect:** **`jmp *.L4(,%rdi,8)`**
  - Start of jump table: **`.L4`**
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address **`.L4 + x*8`**
    - Only for $0 \le \mathbf{x} \le 6$

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```

Code Blocks (x == 1)

```
  switch(x) {
  case 1:         // .L3
        w = y*z;
        break;
    . . .
}
```

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
        w = 1;
merge:
        w += z;
```

Code Blocks (x == 2, x == 3)

```
    long w = 1;
       . . .
    switch(x) {
       . . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
       . . .
    }
```

```
.L5:                        # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx            #  y/z
   jmp      .L6             #  goto merge
.L9:                        # Case 3
   movl     $1, %eax        #  w = 1
.L6:                        # merge:
   addq     %rcx, %rax      #  w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks

(x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl  $1, %eax        #  w = 1
  subq  %rdx, %rax      #  w -= z
  ret
.L8:                    # Default:
  movl  $2, %eax        #  2
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

多判断条件的短路性

- 逻辑与运算

while (sum < 1000) and (count ≤ 24) loop

... { body of loop }

end while;

两个条件必须都成立<==>任一个不成立都退出

```
whileSum:     cmpw $1000, $sum    ; sum < 1000 ?
              jnl  endWhileSum    ; exit if not
              cmpw $24, %cx       ; count <= 24 ?
              jnle  endWhileSum   ; exit if not
              . . .               ; body of loop . .
              jmp whileSum        ; go check condition again
endWhileSum:
```

多判断条件的短路性

- 逻辑或运算

```
while (sum < 1000) or (flag=1) loop
        ... { body of loop }
end while;
```

```
whileSum:   cmpw $1000, $sum    ; sum < 1000 ?
            jl  body            ; execute body if so
            cmpb $1, %dh                ; flag=1 ?
            jne  endWhileSum    ; exit if not
body:       . . .               ; body of loop . .
            ....
            jmp  whileSum       ; go check condition again
endWhileSum:
```

Loop 指令实现 for 循环

- LOOP label
  - 利用 ECX 计数器自动减1，方便实现计数循环的程序结构。
  - ECX ← ECX－1 ;dec ecx
  - 如果 ECX≠0，转标号 label 继续循环;jnz label
  - 否则继续执行循环指令下面的语句
  - 操作数 label 采用相对短转移寻址方式，及向后128个字节或向前127个字节。

使用“loop”的80x86汇编代码

```
for  count := 20 downto 1   loop
        ... { body of loop }
end for
```

```
                movl $20, %ecx          ; number of iterations
forCount:       . . .                   ; body of loop
                . . .
                loop forCount           ; repeat body 20 times
```

```
                movl $number, %ecx      ; number of iterations
forIndex:       . . .                   ; body of loop
                . . .
                loop forIndex           ; repeat body number times
```

可靠的 for 循环

- 当ecx中的值为0时，循环不会执行

```
        movl $number, %ecx      ; number of iterations
        cmpl $0, %ecx           ; number = 0 ?
        jle  endFor             ; skip loop if number = 0
forIndex: . . .                 ; body of loop
        . . .
        loop forIndex           ; repeat body number times
endFor:
```

```
        movl $number, %ecx      ; number of iterations
        jecxz  endFor           ; skip loop if number = 0
forIndex: . . .                 ; body of loop
        . . .
        loop forIndex           ; repeat body number times
endFor:
```

课堂练习

填写C语言代码：

```
Long test (long x, long y) {
    long val = ____8x____;
    if (____y>0____) {
        if (____x<y____) {
            val = ____y-x____;
        else
            val = ____y&x____;
    } else if (____y<=-2____)
        val = ____x+y____;
    return val;
}
```

```
# x @ %rdi, y @ %rsi
Test:
  leaq    0(,%rdi,8), %rax
  testq   %rsi, %rsi
  jle     .L2
  movq    %rsi, %rax
  subq    %rdi, %rax
  movq    %rdi, %rdx
  andq    %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge %rdx, %rax
.L2:
  addq    %rsi, %rdi
  cmpq    $-2, %rsi
  cmovle %rdi, %rax
```

练习答案

```
Long test (long x, long y) {
    long val = ____8x____;
    if (___y>0____) {
        if (___x<y____) {
            val = ___y-x_____;
        else
            val = ___x&y_____;
    } else if (___y<=-2____)
        val = __x+y_____;
    return val;
}
```

```
# x @ %rdi, y @ %rsi
Test:
  leaq    0(,%rdi,8), %rax
  testq   %rsi, %rsi
  jle     .L2
  movq    %rsi, %rax
  subq    %rdi, %rax
  movq    %rdi, %rdx
  andq    %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge %rdx, %rax
.L2:
  addq    %rsi, %rdi
  cmpq    $-2, %rsi
  cmovle %rdi, %rax
  ret
```

课堂练习

程序填空，变量映射关系，并解释函数功能

```
Int fun_a(unsigned x) {
  int val =
                x     ) {

    x>>=1
  }
  return val&0x01;
}
```

- x@$ebp+8

```
  Movl 8(%ebp), %edx
  Movl $0, %eax
  Testl %edx, %edx
  Je .L7
.L10:
  xorl %edx, %eax
  shrl %edx  ;逻辑右移1位
  jne .L10
.L7:
  andl $1, %eax
```

课堂练习

程序填空，变量映射关系，并解释函数功能

```
Int fun_a(unsigned x) {
  int val = 0;
  while (    x     ) {
    val ^= x;
    x>>=1
  }
  return val&0x01;
}
```

- x@$ebp+8

```
  Movl 8(%ebp), %edx
  Movl $0, %eax
  Testl %edx, %edx
  Je .L7
.L10:
  xorl %edx, %eax
  shrl %edx  ;逻辑右移1位
  jne .L10
.L7:
  andl $1, %eax
```

课堂练习

- 函数fun_b(unsigned long x) {
  - Long val = 0;
  - Long i;
  - For (… ; … ; …) {
    - …
  - }
  - Return val;
- }
- Gcc生成的汇编代码如右所示。
- 完成下面工作：
  - A. 根据汇编代码，填写C代码缺失的部分；
  - B. 解释循环前为什么没有初始测试，也没有初始跳转到循环内部的测试部分；
  - C. 用自然语言描述这个函数的功能

```
# x in %rdi
1 func_b:
2     Movl $64, %edx
3     movl $0, %eax
4   .L10:
5     movq %rdi, %rcx
6     andl  $1, %ecx
7     addq %rax, %rax
8     orq % rcx, %rax
9     shrq %rdi
10    subq $1, %rdx
11    jne .L10
12    ret
```

练习答案

- fun_b(unsigned long x) {
  - Long val = 0;
  - Long i;
  - For (i=64; i!=0 ; i--) { // 尽量忠于原意
    - Val = (val << 1) | (x&1)
    - X >>= 1;
  - }
  - Return val;
- }
- 2）因为循环次数是常量
- 3）将x镜像反转

练习3.31

```
long switcher(long a, long b,
long c, long *dest)
{
    long val;
    switch(a) {
    case _____ :
        c = _____;
        /* Fall Through */
    case _____ :
        val = _____;
        break;
    case _____ :
    case _____ :
        val = _____;
        break;
    case _____ :
        val = _____;
        break;
    default:
        val = _____;
}
    *dest=val
}
```

3

```
Switcher:
   cmpq   $7, %rdi
   ja     .L2
   jmp    *.L4(,%rdi,8)
   .section          .rodata
.L7:
   xorq   $15, %ris
   movq %rsi, %rdi
.L3:
   leaq   112(%rdx), %rdi
   jmp    .L6
.L5:
   leaq    (%rdx, %rsi),%rdi
   salq   $2, %rdi
   jmp    .L6
.L2:
   movq     %rsi, %rdi
.L6:
   movq     %rsi,  (%rcx)
   ret
```

2

```
.L4:
  .quad     .L3
  .quad     .L2
  .quad     .L5
  .quad     .L2
  .quad     .L6
  .quad     .L7
  .quad     .L2
  .quad     .L5
```

1

Summarizing

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)

Summary

- Today
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops
  - Switch statements
- Next Time
  - Stack
  - Call / return
  - Procedure call discipline

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
. . .
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
. . .
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
(gdb)
```

Finding Jump Table in Binary

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
```

```
  . . .
  4005f0:       48 89 f0                mov    %rsi,%rax
  4005f3:       48 0f af c2             imul   %rdx,%rax
  4005f7:       c3                      retq
  4005f8:       48 89 f0                mov    %rsi,%rax
  4005fb:       48 99                   cqto
  4005fd:       48 f7 f9                idiv   %rcx
  400600:       eb 05                   jmp    400607 <switch_eg+0x27>
  400602:       b8 01 00 00 00          mov    $0x1,%eax
  400607:       48 01 c8                add    %rcx,%rax
  40060a:       c3                      retq
  40060b:       b8 01 00 00 00          mov    $0x1,%eax
  400610:       48 29 d0                sub    %rdx,%rax
  400613:       c3                      retq
  400614:       b8 02 00 00 00          mov    $0x2,%eax
  400619:       c3                      retq
```