

汇编指令简介

教师：吴锐

计算机科学与技术学院

哈尔滨工业大学

汇编指令简介

- 一、整数算术指令
- 二、布尔和比较指令
- 三、字符串指令

一、整数算术指令

- 加、减、乘、除
- 如何使用移位和循环移位指令移动数字的若干位？
- 为什么计算机能实现大整数的加减法？
- 编译器如何将复杂的表达式分解并翻译成独立的机器语言指令的？
- 在将表达式翻译成汇编语言的时候如何使用运算符优先级和寄存器优化规则？

一、整数算术指令

1. 数据传送、加减指令
2. 移位和循环移位指令
3. 移位和循环移位的应用
4. 乘法和除法指令
5. 扩展加法和减法
6. 十进制调整指令

1.1 数据传送、加减指令

MOV指令

MOVZX指令

MOVSX指令

LAHF指令

SAHF指令

XCHG指令

LAHF、SAHF指令

■ LAHF (load status flags into AH)

- 将EFLAGS寄存器的低字节拷贝至AH，被拷贝的标志包括：符号标志SF、零标志ZF、辅助进位标志AC、奇偶标志PF和进位标志CF。

■ SAHF (store AH into status flags)

- 拷贝AH寄存器的值至EFLAGS的低字节
- 用如下指令恢复刚才保存在变量中的标志:

```
.data
saveflags : .byte 0
.code
lahf
mov %ah, saveflags
.....
mov saveflags, %ah
sahf
```

XCHG指令

■ XCHG指令:交换两个操作数的内容

xchg reg, reg

xchg reg, mem

xchg mem, reg

- 操作数规则遵循与MOV同样的规则。

xchg %ax, %bx

xchg %ah, %al

xchg var1, %bx

xchg %eax, %ebx

mov var1, %ax

xchg %ax, var2

mov %ax, var1

交换两个内存操作数
需使用寄存器

加法和减法指令

- INC、DEC
- ADD
- SUB
- ADC
- SBB
- NEG

INC和DEC指令

- INC (DEC)指令从操作数中加1(减1)

`inc reg/mem`

`dec reg/mem`

不影响CF!

.data

varx: .int 0x1234

.text

incw varx

decw varx

incl varx

decl varx

inc %eax

dec %rbx

ADD指令和SUB指令

■ ADD

指令将同尺寸的源操作数和目的操作数相加。

add源操作数, 目的操作数

- 加法操作并不改变源操作数，结果存储在目的操作数中。
- 影响的标志：进位标志CF、零标志ZF、符号标志SF、溢出标志OF、辅助进位标志AF和奇偶标志PF(结果低8位中，数值1 的个数是否为偶数)。

ADD指令和SUB指令

例如：

```
.data
```

```
var1: .int 0x10000
```

```
var2: .int 0x20000
```

```
.text
```

```
mov var1, %eax
```

```
add var2, %eax,
```

; eax=30000h,var1=10000h,var2=20000h

; CF=0,SF=0,ZF=0,OF=0

ADD指令和SUB指令

■ SUB

将源操作数从目的操作数中减掉。

sub 源操作数, 目的操作数

影响的标志：**CF、ZF、SF、OF、AF**和**PF**。

```
.data
```

```
var1 : .int 0x30000
```

```
var2 : .int 0x10000
```

```
.text
```

```
mov var1, %eax
```

```
sub var2, %eax
```

```
; eax=20000h    CF=0,SF=0,ZF=0,OF=0
```

ADC、SBB

■ ADC 带进位的加法

ADC src, dst

src+dst+CF → dst

■ SBB 带借位的减法

SBB src, dst

dst-src-CF → dst

example of using the ADC instruction

```
.section .data
```

```
data1: .quad 7252051615
```

```
data2: .quad 5732348928
```

```
output: .asciz "The result is %qd\n"
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
movl data1, %ebx
```

```
movl data1+4, %eax
```

```
movl data2, %edx
```

```
movl data2+4, %ecx
```

```
addl %ebx, %edx
```

```
adcl %eax, %ecx
```

```
pushl %ecx
```

```
pushl %edx
```

```
push $output
```

```
call printf
```

```
addl $12, %esp
```

```
pushl $0
```

```
call exit
```

NEG指令

■ NEG

将数字转换为对应的二进制补码,从而求得其相反数。
影响的标志位同ADD指令。

neg reg

neg mem

加减运算示例

;Rval = -Xval + (Yval - Zval)

.data

Rval: .int 0

Xval: .int 26

Yval: .int 30

Zval: .int 40

.text

mov Xval,%eax

neg %eax

mov Yval,%ebx

sub Zval,%ebx

add %ebx,%eax

mov %eax,Rval #eax=0xff ff ff dc Rval=-36

1.2 移位和循环移位指令

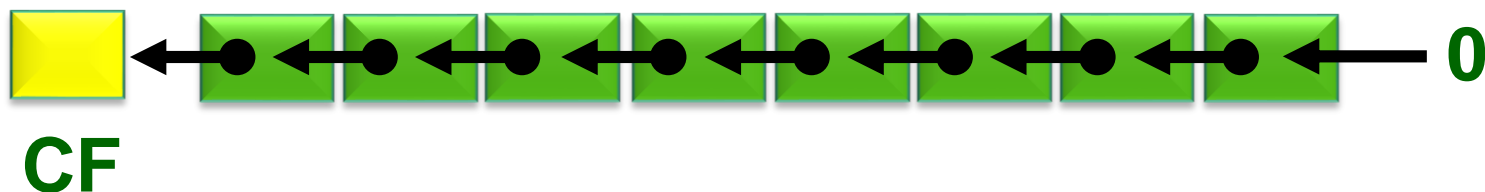
- SHL/SHR: 逻辑左/右移位
- SAL/SAR: 算术左/右移位
- ROL/ROR: 循环左/右移位
- RCL/RCR: 带进位CF的循环左/右移位
- SHLD/SHRD: 双精度左/右移位

1.2.1 SHL指令

- SHL (Shift left) : 对目的操作数执行逻辑左移操作，低位填0，移出的最高位送CF

SHL 移位位数, 目的操作数

格式: SHL imm8/CL, reg/mem



```
mov bl, 8fh
shl bl, 1
```

快速乘法:

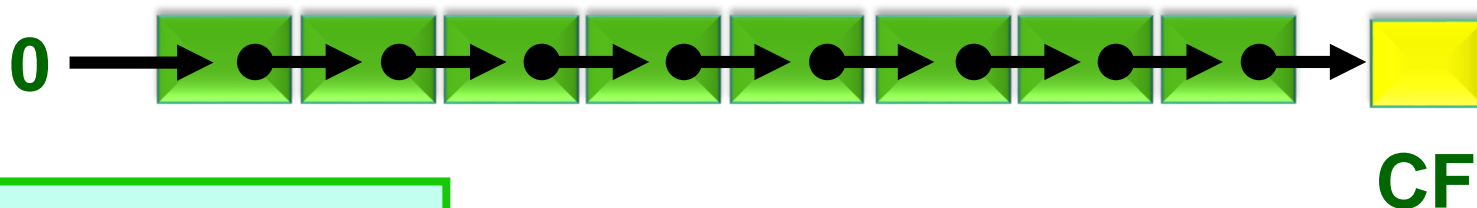
左移 n 位等价于乘以 2^n

1.2.2 SHR指令

- SHR (shift right) : 对目的操作数执行逻辑右移操作, 移出的数据位以0填充, 最低位被送到CF中

SHR 移位位数, 目的操作数

格式: SHR imm8/CL, reg/mem



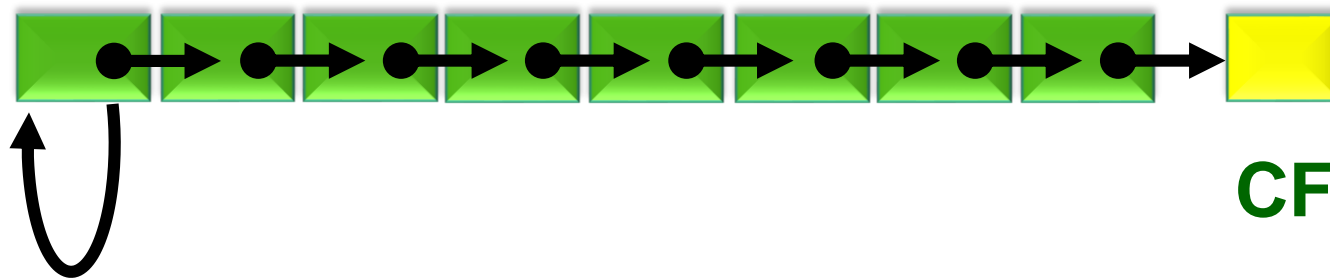
```
mov al, 0d0h  
shr al, 1
```

无符号数快速除法:

右移 n 位等价于除以 2^n

1.2.3 SAL和SAR指令

- SAL指令与SHL指令等价；
- SAR指令：用最高位填充空出的位，最低位拷贝至CF



1.2.3 SAL和SAR指令

■ 比较SAR与SHR:

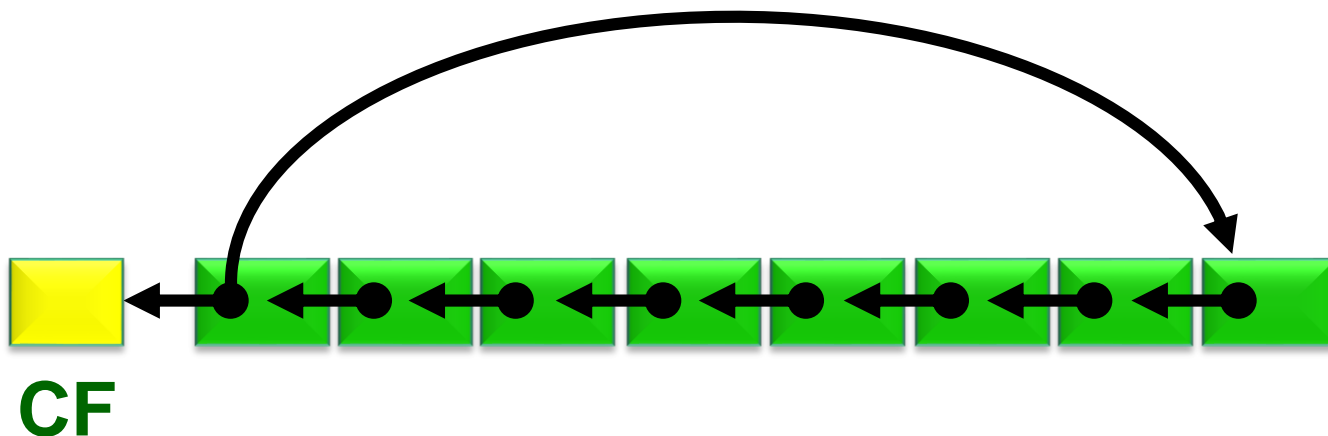
```
MOV $0xF0, %AL  
SAR $1, %AL
```

```
MOV $0xF0, %AL  
SHR $1, %AL
```

快速除法（有符号）

1.2.4 ROL指令

- ROL (rotate left) 指令向左移动，并将最高位同时拷贝到CF和最低位中；



1.2.4 ROL指令

```
MOV $0x40, %AL
```

```
ROL $1, %AL,
```

```
ROL $1, %AL
```

```
ROL $1, %AL
```

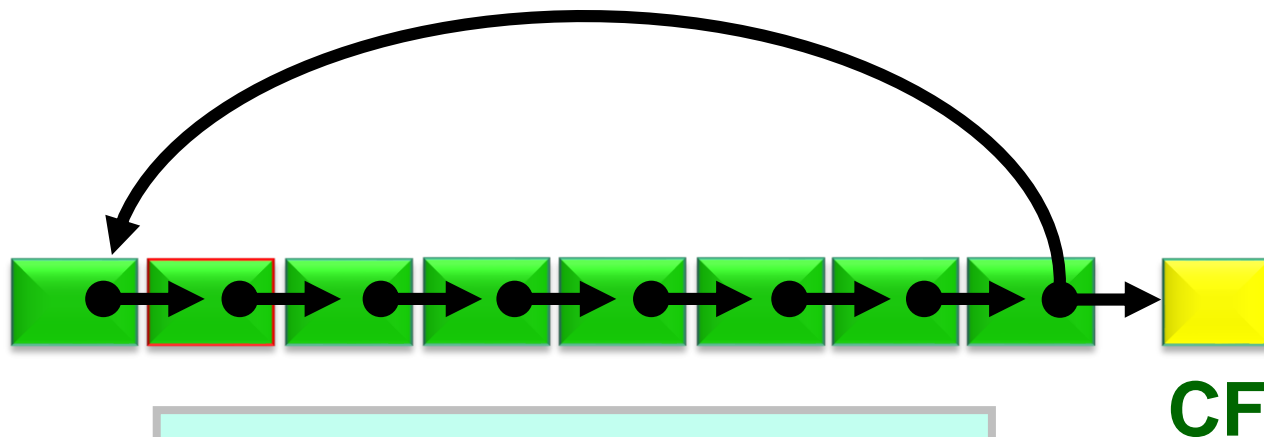
```
MOV $0x26, %AL
```

```
ROL $4, %AL
```

交换一个字节的
高4位和低4位！

1.2.5 ROR指令

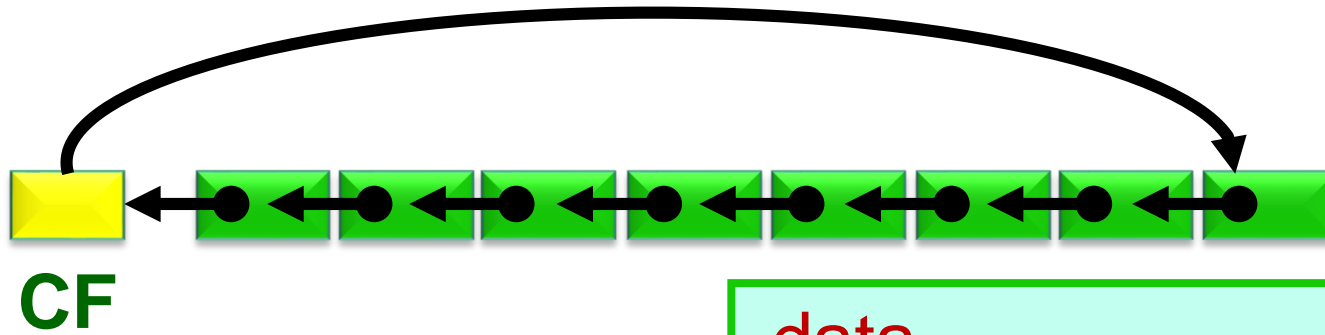
- ROR (rotate right) 指令向右移动, 并将最低位同时拷贝到CF和最高位中;



```
MOV $01, %AL  
ROR $1, %AL  
ROR $1, %AL
```

1.2.6 RCL和RCR指令

- RCL (rotate carry left) 指令按位左移，并将CF拷贝到最低有效位，然后将最高有效位拷贝至CF中；

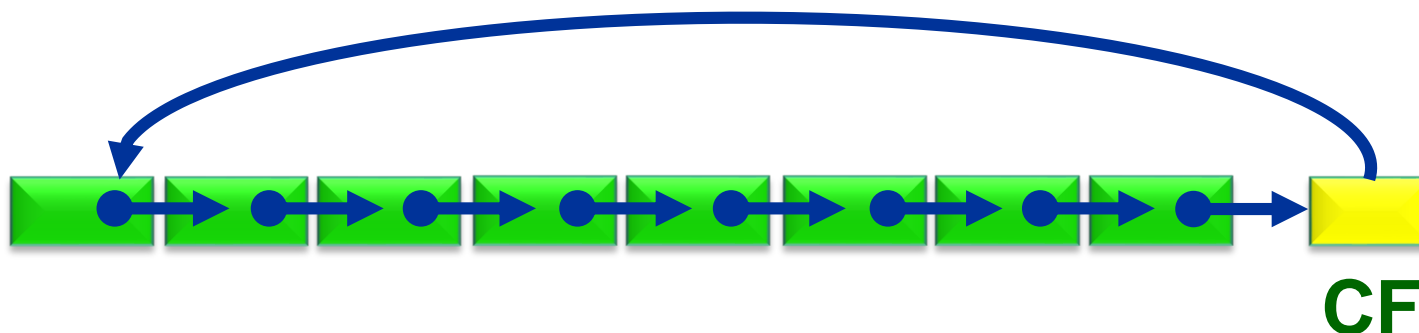


```
CLC  
MOV $0x88, %BL  
RCL $1, %BL  
RCL $1, %BL
```

```
.data  
testval: .byte 0x6a  
  
shrb $1, testval  
jc quit  
rclb $1, testval
```


1.2.6 RCL和RCR指令

- RCR (rotate carry right) 指令按位右移，并将CF拷贝到最高有效位，然后将最低有效位拷贝至CF中；



STC

MOV \$0x10, %AH

RCR \$1, %AH

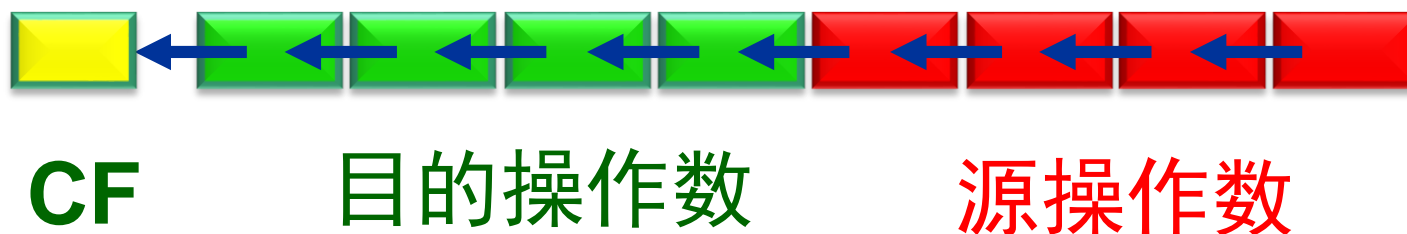
shr/shl/sal/sar/rol/ror/rcl/rcr %rax #移动1位

1.2.7 SHLD/SHRD指令

■ SHLD (shift left double) 双精度左移

将目的操作数左移指定的位数；左移空出来的位用源操作数的高位来填充。

SHLD 移位位数，源操作数，目的操作数

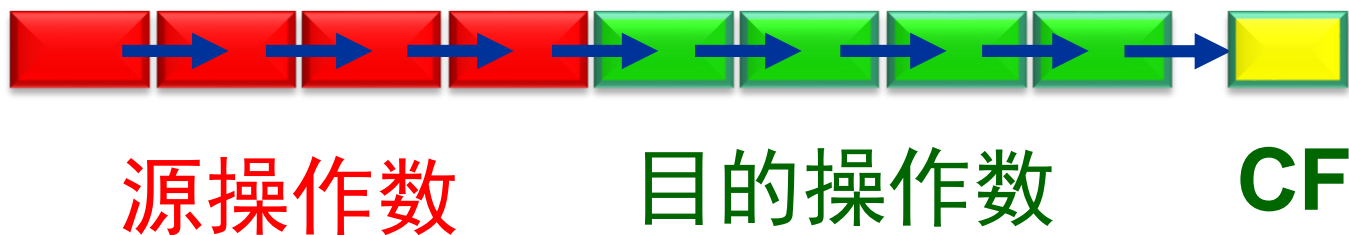


1.2.7 SHLD/SHRD指令

■ SHRD (shift right double) 双精度右移

将目的操作数右移指定的位数；右移空出来的位用源操作数的低位来填充。

SHRD 目的操作数，源操作数，移位位数



1.2.7 SHLD/SHRD指令

指令格式:

SHLD/SHRD imm8/CL, reg16, reg16

SHLD/SHRD imm8/CL, reg32, reg32

SHLD/SHRD imm8/CL, reg64, reg64

SHLD/SHRD imm8/CL, reg16, mem16

SHLD/SHRD imm8/CL, reg32, mem32

SHLD/SHRD imm8/CL, reg64, mem64

注意: 第二个操作数必须是寄存器,
不能是mem

1.2.7 SHLD/SHRD指令的例子

#varx: .int 0x12

mov \$0x12345678, %eax # eax = 0x12345678

mov \$0x98765432,%ecx #

shld \$4, %ecx,%eax # eax = 0x23456789




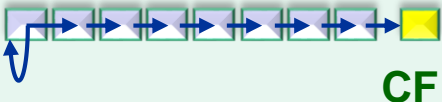

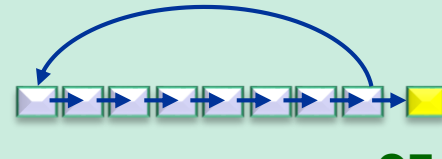
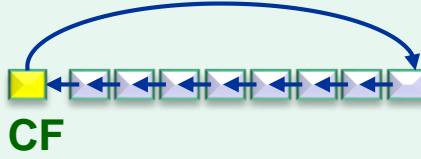
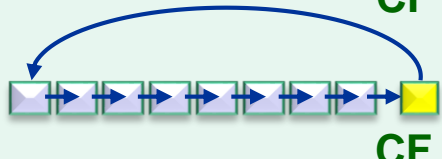


shrd \$4,%ecx,%eax # eax = 0x22345678

mov \$4,%cl

movl \$0x01234567,varx # varx =0x01234567

shrd %cl,%eax,varx # varx =0x80123456

移位指令汇总

移位类型	左移	右移
逻辑移位	SHL 	SHR 
算术移位	SAL 	SAR 
循环移位	ROL 	ROR 
带进位循环移位	RCL 	RCR 
双精度移位	SHLD 	SHRD 

所有指令都影响CF和OF

1.3 移位和循环移位的应用

- 多双字移位
- 二进制乘法
- 显示二进制位
- 分离位串

数组移位：右移1位(intel 格式)

```
.data
    arraysize = 10
    array dword arraysize dup(76543210h)
.code
main PROC
    mov esi, (arraysize-1)*(type array)
    mov ecx, arraysize
    clc
    lahf
```

```
L:
    sahf
    rcr array[esi],1
    lahf
    sub esi,type array
    loop L
exit
main ENDP
END main
```

将上述intel格式代码改写成ATT格式！

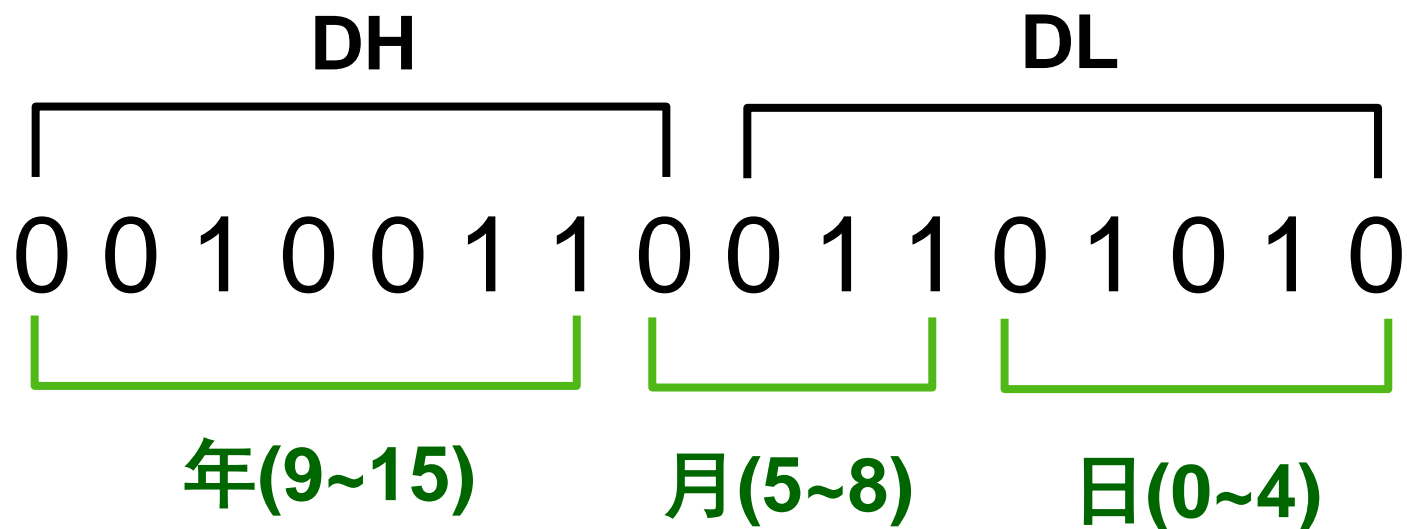
二进制乘法

- 为了应用SHL，将任意二进制乘数分解成2的幂的和：

$$\begin{aligned} X * 36 &= X * 32 + X * 4 \\ &= X * 2^5 + X * 2^2 \end{aligned}$$

- X、Y是word或dword类型，分别如何实现X*Y？

分离位串



- 将要提取的位移位到寄存器的最低部分
- 清除不相关的位。

1.4 乘法和除法指令

- 无符号乘法指令MUL
- 有符号乘法指令IMUL
- 无符号除法指令DIV
- 有符号整数除法指令IDIV
- 算术表达式的实现

MUL指令

■ MUL指令：无符号乘法指令、单操作数

■ 指令格式： **MUL r/m8 ;AX = AL * r/m8**

MUL r/m16 ;DX:AX = AX * r/m16

MUL r/m32 ;EDX:EAX = EAX * r/m32

MUL r/m32 ;RDX:RAX = RAX * r/m64

被乘数、积由乘数隐含指定：

乘数	被乘数	积
r/m8	AL	AX
r/m16	AX	DX:AX
r/m32	EAX	EDX:EAX
r/m64	RAX	RDX:RAX

操作数不可以是立即数！

如果积的高半部分不为0，则CF=OF=1

MUL指令

```
mov $0x5,%al  
mov $0x20,%bl  
mul %bl
```

- 被乘数是AL，乘积被放入AX中
- 根据乘积中高半部分是否为0，设置或清除CF位和OF位
- 处理无符号数，只需关心CF标志位

AX= 0x00A0

CF = 0、OF=0

MUL指令

```
.data  
val1: .short 0x2000  
val2: .short 0x0100  
.text  
mov val1, %ax  
mulw val2
```

- 被乘数是AX，乘积被放入DX: AX中；
- 根据乘积中高半部分是否为0，设置或清除CF位和OF位；
- 处理无符号数，只需关心CF标志位。

AX=0x0000
DX=0x0020
CF = 1、OF=1

MUL指令

```
mov 0x12345, %eax  
mov 0x1000,%ebx  
mul %ebx
```

EAX = 0x12345000
EDX = 0x00000000
CF = 0 OF=0

1. 被乘数是EAX，乘积被放入EDX：EAX中；
2. 根据乘积中高半部分是否为0，设置或清除CF位和OF位；
3. 因为处理的是无符号数，所以只关心CF标志位；

IMUL指令

■ IMUL指令：有符号乘法指令

➤ IMUL指令的单操作数格式

IMUL r/m8 ;AX = AL*r/m8 byte

IMUL r/m16 ;DX:AX = AX*r/m16 word

IMUL r/m32 ;EDX:EAX=EAX* r/m32 double word

IMUL r/m64 ;RDX:RAX=RAX* r/m64 double word

操作数不可以是立即数！

有效位进位到结果的上半部分（低半部分全是数值位），CF 与 OF 标志设置为 1

IMUL指令的多操作数形式

- IMUL指令的多操作数格式

- 双操作数：2个操作数的积保存到第2个操作数

IMUL r16/m16/imm8/imm16, r16

IMUL r32/m32/imm8/imm32, r32

- 三操作数：前2个操作数的积保存到第3个操作数

IMUL imm8/16, r16/m16, r16

IMUL imm8/32, r32/m32, r32

第三个操作数必须是寄存器



会根据目的操作数的大小裁减乘积



如果有效位丢失，则设置CF和OF



需要检查CF、OF，以防结果溢出

IMUL指令

```
mov $48,%al  
mov $4,%bl  
imul %bl
```

AX=0x00C0

OF = 1

1. 被乘数是AL，乘积被放入AX中
2. 根据乘积中高半部分是不是低半部分的扩展，设置或清除CF位和OF位
3. 因为处理的是有符号数，所以关心OF标志位

AX=0xFFFF0

OF = 0

```
mov $-4,%al  
mov $4,%bl  
imul %bl
```

IMUL指令

```
mov  $-16,%ax  
mov  $2,%bx  
imul %ax,%bx
```

BX=-32

OF = 0

1. 被乘数是BX，乘积被放入BX中
2. 根据乘积中的有效位是否被裁减，设置或清除CF位和OF位
3. 因为处理的是有符号数，所以关心OF标志位

```
mov  $-32000, %ax  
imul $2,%ax
```

AX=0x0600

OF = 1

DIV指令

■ DIV指令：无符号除法指令

- 单操作数指令
- 执行8、16、32、64位无符号除法
- 被除数、商以及余数都由除数的大小决定：

除数	被除数	商	余数
r/m8	AX	AL	AH
r/m16	DX:AX	AX	DX
r/m32	EDX:EAX	EAX	EDX
r/m64	RDX:RAX	RAX	RDX

DIV指令

```
mov $0x0083,%ax  
mov $2, %bl  
div  %bl
```

1. 除数是8位的，那么被除数就应该放入16位的AX中
2. 商被放入AL中，余数被放入AH中

AL = 0x41

AH = 0x01

DIV指令

```
mov $0,%dx  
mov $0x8003,%ax  
mov $0x100, %cx  
div  %cx
```

1. 除数是16位时，那么被除数的高16位放入DX中，低16位放入AX中
2. 商被放入AX中，余数被放入DX中

AX = 0x0080

DX = 0x0003

DIV指令

```
.data
dividend: .quad 0x80030020h
divisor:  .int 0x100h
.text
    mov    dividend+4,%edx
    mov    dividend,  %eax
    divl   divisor
```

EAX = 0x800300

EDX = 0x20

有符号数除法

■ IDIV指令

■ CBW、CWD、CDQ、CQO指令

■ 除法溢出

IDIV指令、整数符号扩展指令

- IDIV指令：有符号除法指令，指令格式同DIV，但需要对被除数的符号进行扩展：
 - 当执行8位除法指令前必须把AL中的被除数符号扩展到AH中（用CBW指令）；
 - 当执行16位除法指令前必须把AX中的被除数符号扩展到DX中（用CWD指令）；
 - 当执行32位除法指令前必须把EAX中的被除数符号扩展到EDX中（用CDQ指令）；
 - 64位除法？用CQO指令

整数符号扩展指令

■ CBW、CWD、CDQ、CQO指令

用于整数符号扩展：

- **CBW**:将AL中的符号位扩展到AH中
- **CWD**:将AX中的符号位扩展到DX中
- **CDQ**:将EAX中的符号位扩展到EDX中
- **CQO** :将RAX中的符号位扩展到RDX中

8位有符号除法

```
.data
byteVal: .byte -48
.text
movb byteVal, %al
cbw
mov $5,%bl
idiv %bl
```

1. 必须将被除数的符号位从AL扩展到AH
2. 商被放入AL中，余数被放入AH中

AL = 0xF7 = -9

AH = 0xFD = -3

16位有符号除法

```
.data  
wordVal: .short -5000  
.text  
    mov wordVal, %ax  
    cwd  
    mov $256, %bx  
    idiv %bx
```

1. 必须将被除数的符号位从AX扩展到DX
2. 商被放入AX中，余数被放入DX中

AX = FFEDH = -19

DX = FF78H = -136

32位有符号除法

.data

Val: .int -5000

.text

movl Val, %eax

cdq

mov \$256, %ebx

idiv %ebx

1. 必须将被除数的符号位从EAX扩展到EDX
2. 商被放入EAX中，余数被放入EDX中

EAX = 0xFFFFFFFFED = -19

EDX = 0xFFFFFFFF78 = -136

.data

byteVal: .byte -65

wordVal: .int -65

dwordVal: .int -65

.text

mov \$14, %ecx

mov byteVal, %al

cbw

idiv %ch

mov wordVal, %ax

cwd

idiv %cx

mov dwordVal, %eax

cdq

idiv %ecx

除法溢出

- ❖ DIV/IDIV执行后，所有算术状态标志均不确定！
- ❖ 除法的商太大，目的操作数无法容纳→除法溢出；
- 除法溢出→ CPU触发中断，终止程序运行。

`mov $0x1000, %ax`

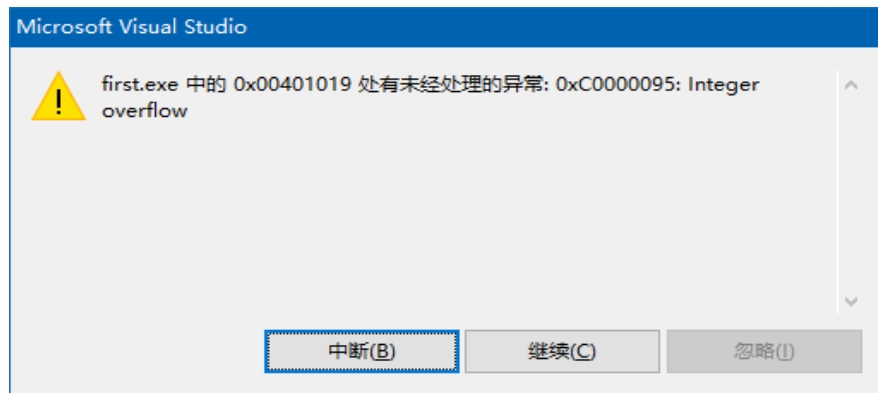
`mov $0x10, %bl`

`div %bl # AL 无法容纳结果100h`

除0运算。

如何防止？

用更多位数的除法、检查除数确定不为0



```
(gdb) s
Program received signal SIGFPE, Arithmetic exception.
_start () at try64.s:35
3: /x $rbx = 0x10
2: /x $rax = 0x1000
```

算术表达式的实现

下列内存操作数均为32位整数：

- $\text{var4} = (\text{var1} + \text{var2}) * \text{var3}$
- $\text{var4} = (\text{var1} * 5) / (\text{var2} - 3)$
- $\text{var4} = (\text{var1} * -5) / (-\text{var2} \% \text{var3})$

1.6 ASCII和非压缩十进制调整指令

- 十进制数调整指令对二进制运算的结果进行十进制调整，以得到十进制的运算结果
- 分成非压缩BCD码和压缩BCD码调整

□ 非压缩BCD码用8个二进制位表示一个十进制位，只用低4个二进制位表示一个十进制位0~9，高4位任意，通常默认为0

□ 压缩BCD码就是通常的8421码；它用4个二进制位表示一个十进制位，一个字节可以表示两个十进制位，即00~99

BCD码 (Binary Coded Decimal)

- 二进制编码的十进制数：一位十进制数用4位二进制编码来表示
- 压缩BCD码和非压缩BCD码的调整运算

真值	8	64	96
二进制编码	0x08	0x 40	0x 60
压缩BCD码	0x08	0x 64	0x 96
非压缩BCD码	0x08	0x 0604	0x 0906

非压缩BCD码加法调整指令——AAA

■ AAA(ASCII adjust **after addition**)

操作内容：

$AL \leftarrow$ 将AL中的加法和调整为非压缩的BCD码(AL高4位清0)；

$AH \leftarrow AH +$ 调整产生的进位

用法：

该指令跟在以al为目的操作数的add或adc指令后，如果调整产生进位， $CF = AF = 1$, 否则 $CF = AF = 0$.

非压缩BCD码加、减调整指令

(ADD AL, i8/r8/m8)

(ADC AL, i8/r8/m8)

AAA

#AL←将AL的加和调整为非压缩BCD码

#AH←AH+调整的进位

(SUB AL, i8/r8/m8)

(SBB AL, i8/r8/m8)

AAS

#AL←将AL的减差调整为非压缩BCD码

#AH←AH-调整的借位

- ❖ 使用AAA或AAS指令前，应先执行以AL为目的操作数的加法或减法指令
- ❖ AAA和AAS指令在调整中产生了进位或借位，则AH要加上进位或减去借位，同时CF=AF=1，否则CF=AF=0；它们对其他标志无定义

非压缩BCD加法调整指令——AAA

mov \$0x0608, %ax

#ax=0608h , 非压缩BCD码表示真值68

mov \$9, %bl

#bl=09h , 非压缩BCD码表示真值9

add %bl, %al

#二进制加法 : al=0x08+0x09=0x11

aaa

#十进制调整 : ax=0x0707

#实现非压缩BCD码加法 : 68 + 9 = 77

非压缩BCD减法调整指令——AAS

mov \$0608, %ax

ax=0x0608, 非压缩BCD码表示真值68

mov \$09, %bl

bl=0x09, 非压缩BCD码表示真值9

sub %bl, %al

#二进制减法: $al = 0x08 - 0x09 = 0xff$

aas

#十进制调整: $ax = 0x0509$

#实现非压缩BCD码减法: $68 - 9 = 59$

非压缩BCD码乘、除调整指令

(**MUL** r8/m8)

AAM

AX ← 将AX的乘积调整为非压缩BCD码

AAD

AX ← 将AX中非压缩BCD码扩展成二进制数

(**DIV** r8/m8)

- ❖ AAM指令跟在字节乘MUL之后，将乘积调整为非压缩BCD码
- ❖ AAD指令跟在字节除DIV之前，先将非压缩BCD码的被除数调整为二进制数
- ❖ AAM和AAD指令根据结果设置SF、ZF和PF，但对OF、CF和AF无定义

非压缩BCD乘法调整指令—— AAM

mov \$0x0608,%ax

ax=0x0608, 非压缩BCD码表示真值68

mov \$0x09,%bl

bl=0x09, 非压缩BCD码表示真值9

mul %bl

#二进制乘法: $al=0x08 \times 0x09=0x0048$

aam

#十进制调整: ax=0x0702

#实现非压缩BCD码乘法: $8 \times 9 = 72$

非压缩BCD除法调整指令—— AAD

mov \$0x0608, %ax

#ax=0x0608, 非压缩BCD码表示真值68

mov \$09, %bl

bl=0x09, 非压缩BCD码表示真值9

aad

#二进制扩展: ax=68=0x0044

div %bl

#除法运算: 商al=0x07, 余数ah=0x05

#实现非压缩BCD码初法:

$$68 \div 9 = 7 \text{ (余5)}$$



1.7 压缩BCD码加、减调整指令

(*ADD AL, i8/r8/m8*)

(*ADC AL, i8/r8/m8*)

DAA

AL ← 将AL的加和调整为压缩BCD码

(*SUB AL, i8/r8/m8*)

(*SBB AL, i8/r8/m8*)

DAS

AL ← 将AL的减差调整为压缩BCD码

- ❖ 使用DAA或DAS指令前，应先执行以AL为目的操作数的加法或减法指令
- ❖ DAA和DAS指令对OF标志无定义，按结果影响其他标志，例如CF反映压缩BCD码相加或减的进位或借位状态

压缩BCD加法调整指令——DAA

mov \$0x68, %al

#al=0x68 , 压缩BCD码表示真值68

mov \$0x28, %bl

#bl=0x28 , 压缩BCD码表示真值28

add %bl, %al

#二进制加法 : al=0x68+0x28=0x90

daa #十进制调整 : al=0x96

#实现压缩BCD码加法 : 68 + 28 = 96

压缩BCD减法调整指令—— DAS

mov \$0x68, %al

#al=0x68 , 压缩BCD码表示真值68

mov \$0x28, %bl

bl=0x28 , 压缩BCD码表示真值28

sub %bl,%al

#二进制减法 : al=0x68-0x28=0x40

das **#十进制调整 : al=0x40**

#实现压缩BCD码加法 : 68-28 = 40

压缩BCD减法调整指令—— DAS

```
mov $0x1234, %ax
```

```
mov $0x4612,%bx
```

```
sub %bl,%al
```

```
das          # 34-12 = 22 , CF = 0
```

```
xchg %al, %ah
```

```
sbb %bh, %al
```




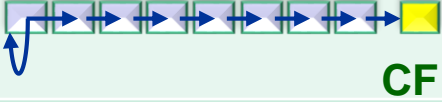
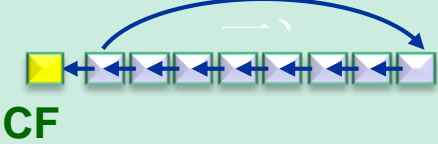
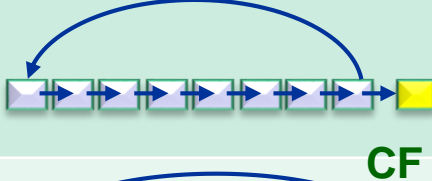
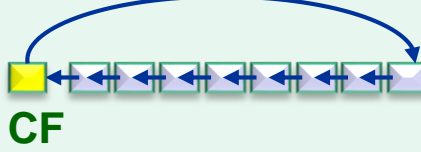
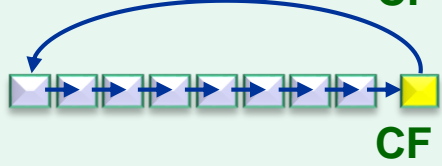


```
das          # 12-46 = 66 , CF=1
```

```
xchg %al, %ah # 11234 - 4612 = 6622
```

加、减、乘、除指令汇总

	加法	减法	乘法	除法
无符号 二进制数	ADD/ADC	SUB/SBB	MUL	DIV
有符号 二进制数	ADD/ADC	SUB/SBB	IMUL	CBW、CWD、 CDQ、CQO IDIV
非压缩 BCD码	<i>ADD/ADC</i> AAA	<i>SUB/SBB</i> AAS	<i>MUL</i> AAM	AAD <i>DIV</i>
压缩BCD 码	<i>ADD/ADC</i> DAA	<i>SUB/SBB</i> DAS		

移位指令汇总

移位类型	左移	右移
逻辑移位	SHL 	SHR 
算术移位	SAL 	SAR 
循环移位	ROL 	ROR 
带进位循环移位	RCL 	RCR 
双精度移位	SHLD 	SHRD 

所有指令都影响CF和OF

二、布尔和比较指令

■ 布尔指令

- AND、OR、XOR、NOT
- TEST
- BT、BTC、BTR、BTS

■ 比较指令

- CMP

■ 条件跳转指令

JCond

■ CPU的状态标志：ZF、SF、CF、OF、PF、AF

2.1 布尔指令——AND

■ AND指令

- AND指令在每对操作数的对应数据位之间执行布尔位“与”操作，并将结果存放在目的操作数中：

AND 源操作数, 目的操作数

AND reg/mem/imm, reg

AND reg/imm, mem

✎ 总是使得CF=0、OF=0

✎ 依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——AND

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

```
mov $0x3b, %al  
and $0x0f, %al  
; al=0xb
```

CF=0, OF=0
ZF=0, SF=0, PF=0

2.1 布尔指令——AND

■ 具体应用举例——字符大小写转换

■ 分析：

▪ 'a' (61h) = 01**1**0 0001b

▪ 'A' (41h) = 01**0**0 0001b



and \$20, arrayElem

???

； 保留字符元素的第5位，

； 以确定其是大写还是小写

2.1 布尔指令——OR

- OR指令在每对操作数的对应数据位之间执行布尔位“或”操作，并将结果存放在目的操作数中：

OR 源操作数, 目的操作数

OR reg/mem/imm, reg

OR reg/imm, mem

✎ 总是使得CF=0、OF=0

✎ 依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——OR

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

```
mov  $5, %al  
or   $0x30, %al  
; al=0x35
```

CF=0, OF=0
ZF=0, SF=0, PF=1

2.1 布尔指令——XOR

- XOR指令在每对操作数的对应数据位之间执行布尔位“异或”操作，并将结果存放在目的操作数中：

XOR 源操作数,目的操作数

XOR reg/mem/imm, reg

XOR reg/imm, mem

✎ 总是使得CF=0、OF=0

✎ 依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——XOR

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

```
mov  $0xb5, %al
xor  $0, %al #al=0xb5
mov  $0xcc, %al
xor  $0, %al #al=0xcc
```

CF=0, OF=0

ZF=0, SF=1, PF=0

CF=0, OF=0

ZF=0, SF=1, PF=1

2.1 布尔指令——XOR

■ 利用XOR指令的特性实现简单的数据加密

- 特性：对数值进行两次“异或”操作后其操作效果将被抵消；

$$(X \oplus Y) \oplus Y = X$$

✎ 总是使得CF=0、OF=0

✎ 依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——NOT

- NOT指令将一个操作数的所有数据位取反：

NOT **reg**

NOT **mem**

NOT指令不修改任何状态标志

```
mov $0xf0, %al  
not  %al  
#al = 0x0f=00001111b
```

练习

```
mov $0xf, %al
and $0x3b, %al
mov $0x6d, %al
and $0x4a, %al
mov $0xf, %al
or $0x61, %al
mov $0x94, %al
xor $0x37, %al
```

```
mov $0x7a,% al
not %al
mov $0x3d, % al
and $0x74, % al
mov $0x9b, % al
or $0x35, % al
mov $0x72, % al
xor $0xdc, % al
```

2.1 布尔指令——TEST

- TEST指令：执行隐含的“与”操作，并相应设置标志位。
- TEST指令不修改目的操作数；
- 指令格式和AND指令相同；
- 测试操作数某位或某几位是否被设置时特别有用！
- 当所有测试位都为0时，ZF=1

```
mov  $0x0fe, %al  
test $0x2e, %al
```

2.2 比较指令——CMP

- **CMP指令执行隐含的减法操作:目的操作数-源操作数,并设置标志位,但不保存减法的结果,两个操作数都不会被修改:**

CMP 源操作数,目的操作数

格式与AND相同, 修改OF、SF、ZF、CF、AF和PF

2.2 比较指令——CMP

■ CMP指令后，操作数大小判别方法（利用标志位）

无符号数大小判别

CMP的结果	ZF	CF
目的 < 源	0	1
目的 > 源	0	0
目的 = 源	1	0

有符号数大小判别

CMP的结果	标志
目的 < 源	SF ≠ OF
目的 > 源	SF = OF
目的 = 源	ZF = 1

2.2 比较指令——CMP

- 将下列Intel 格式指令，写成ATT格式

```
mov $5, %ax  
cmp $10, %ax  
mov $1000, %ax  
mov $1000, %cx  
cmp %ax, %cx  
mov $105, %si  
cmp $0, %si
```

```
mov $15, %al  
test $2, %al  
mov $6, %al  
cmp $5, %al  
mov $5, %al  
cmp $7, %al
```

2.3 标志位设置指令

and \$0, %al #ZF=1

or \$1, %al #ZF=0

or \$0x80, %al #SF=1

and \$0x7f, %al #SF=0

stc #CF=1

clc #CF=0

CMC;进位标志取反: $CF \leftarrow \sim CF$

2.4 条件跳转指令

■ 跳转依据

- 特定的标志值
- 操作数之间是否相等，或ECX的值
- 根据比较结果
 - 无符号操作数
 - 有符号操作数

Jcond 目的标号

2.4 条件跳转指令

■ 根据特定标志跳转

- JZ/JNZ

- JC/JNC

- JS/JNS

- JO/JNO

- JP/JNP

例1：

```
mov    status, %al
test   $0x20, %al
jnz    EquipOffline
```

例2：

```
mov    status, %al
test   $0x13, %al
jnz    InputDataByte
```

2.4 条件跳转指令

■ 根据相等比较的跳转指令

- JE/JNE
- JCXZ
- JECXZ

例3:

```
mov    status, %al
test   $0x8c, %al
cmp    $0x8c, %al
je     ResetMachine
```

2.4 条件跳转指令

■ 无符号数比较

- JA/JNA
- JAE/JNAE
- JB/JNB
- JBE/JNBE

```
.data
    v1: .short 1
    v2: .short 2
    v3: .short 3

.text
    mov     v1, %ax
    cmp     v2, %ax
    jbe     L1
    mov     v2, %ax
L1:  cmp     v3, %ax
    jbe     L2
    mov     v3, %ax
L2:
```

2.4 条件跳转指令

■ 有符号数比较

- JG/JNG

- JGE/JNGE

- JL/JNL

- JLE/JNLE

2.4 条件跳转指令

■ 条件跳转的应用

- 两个数的较大值
- 三个数的最小值
- 扫描数组查找特定值
- 字符串加密

2.5 位测试指令

■ BT (bit test) 指令

- 将第一个操作数的第n位拷贝到进位标志CF中

不会被指令所修改！

BT n, 位基(bitBase)

BT r16/imm8, r/m16

BT r32/imm8, r/m32

2.5 位测试指令

■ BTC

- 将第一个操作数的位 n 拷贝到进位标志中，同时将位 n **取反**

■ BTR

- 将第一个操作数的位 n 拷贝到进位标志中，同时将位 n **清零**

■ BTS

- 将第一个操作数的位 n 拷贝到进位标志中，同时将位 n **置位**

2.6 条件循环指令

■ **LOOPZ / LOOPE**

- 在 $ZF=1$ 并且 $ECX \neq 0$ 时循环
- 目标标号据LOOPZ的下一条指令的距离应该在一128到+127字节范围内

■ **LOOPNZ / LOOPNE**

- 在 $ZF=0$ 并且 $ECX \neq 0$ 时循环
- 目标标号据LOOPZ的下一条指令的距离应该在一128到+127字节范围内

三、字符串操作指令

- REP、REPZ/REPE、REPNZ/REPNE
- MOVSB、MOVSW、MOVSD、MOVSQ
- CMPSB、CMPSW、CMPSD、CMPSQ
- SCASB、SCASW、SCASD、SCASQ
- LODSB、LODSW、LODSD、LODSQ
- STOSB、STOSW、STOSD、STOSQ

并不局限于字符串，关键看指令执行的操作！

三、字符串操作指令

- 数据串(数组): 以字节、字和双字为单位的多个数据存放在连续的主存区域中
- 源操作数: 允许段跨越: DS:[ESI]
- 目的操作数: 不允许段跨越: ES:[EDI]
- 每执行一次串操作: ESI和EDI自动 $\pm 1/2/4/8$
 - 以字节为单位(用B结尾)操作: 地址指针 ± 1
 - 以字为单位(用W结尾)操作: 地址指针 ± 2
 - 以双字为单位(用D结尾)操作: 地址指针 ± 4
 - 以4字为单位(用Q结尾)操作: 地址指针 ± 8
 - DF=0(执行CLD指令): 地址指针增加(+)
 - DF=1(执行STD指令): 地址指针减小(-)

三、字符串操作指令

■ 串传送指令

MOVSB|**MOV**SW|**MOV**SD|**MOV**SQ

;串传送: $ES:[EDI] \leftarrow DS:[ESI]$

$[RDI] \leftarrow [RSI]$

;然后: $ESI \leftarrow ESI \pm 1/2/4, EDI \leftarrow EDI \pm 1/2/4$

$RSI \leftarrow RSI \pm 8, RDI \leftarrow RDI \pm 8$

STOSB|**STO**SW|**STO**SD

;串存储: $ES:[EDI] \leftarrow AL/AX/EAX$

$[RDI] \leftarrow RAX$

;然后: $EDI \leftarrow EDI \pm 1/2/4$

$RDI \leftarrow RDI \pm 8$

LODSB|**LOD**SW|**LOD**SD

;串读取: $AL/AX/EAX \leftarrow DS:[ESI]$

$RAX \leftarrow [RSI]$

;然后: $ESI \leftarrow ESI \pm 1/2/4$

$RSI \leftarrow RSI \pm 8$

REP

;执行一次串指令, ECX减1; 直到 $ECX=0$

三、字符串操作指令

■ 串检测指令

CMPSB | CMPSW | CMPSD | CMPSQ

;串比较: DS:[ESI] - ES:[EDI] [RSI] - [RDI]

;然后: $ESI \leftarrow ESI \pm 1/2/4$, $EDI \leftarrow EDI \pm 1/2/4$ $RSI \leftarrow RSI \pm 8$, $RDI \leftarrow RDI \pm 8$

SCASB | SCASW | SCASD | SCASQ

;串扫描: AL/AX/EAX - ES:[EDI] RAX - [RDI]

;然后: $EDI \leftarrow EDI \pm 1/2/4$ $RDI \leftarrow RDI \pm 8$

REPE | REPZ

;执行一次串指令, ECX减1; 直到ECX=0或ZF=0

REPNE | REPNZ

;执行一次串指令, ECX减1; 直到ECX=0或ZF=1

① 串拷贝指令

.data

string1 : .asciz "this is a example\r\n"

count = . - string1 # 字符串长度

.bss

.comm string2, count

.text

cld

mov \$string1, **%rsi # esi**

lea string2, **%rdi # edi**

mov \$count, **%ecx**

rep movsb

1. cld/std设定每次执行 movsb指令时, rdi/edi, rsi/esi变化的方向;
2. rep: 首先检测ecx>0?, 如=0则执行下一条指令, 否则ecx--, 执行 movsb
3. movsb: 会自动将rdi / edi、rsi/esi值增加/减小

② 串比较指令

.data

source: .int 0x5634

target: .int 0x1278

.text

lea source,%rsi # esi

lea target , %rdi # edi

cmpsb

ja L1

jmp L2

Cmps指令的显示格式:

cmps dword ptr [esi],[edi]

只比较ESI、EDI 指向的内存值

常见错误:

lea source,%eax

lea target , %ebx

cmpsd %eax,%ebx

换成cmpsw 将如何?

比较多个双字:

mov \$count, %ecx

cld

repe cmpsd

③ 串扫描指令

.data

string1 : .ascii "ABCDEFGH"

Len : . - string1

.text

lea string1, **%rdi** **#%edi**

mov '\$F', %al

mov \$Len, %ecx

cld

repne **e** scasb

jnz quit **#一直到最后都没有找到**

dec **%rdi** **#%edi** **#找到, rdi/edi需倒退一位(b)**

将AL/AX/EAX的值同内存中的字节、字或双字比较，目标内存由edi寻址