

# **Lecture 3. Logic operations. Branching and loops. Basic addressing modes.**

Dmitry Zaitsev

<http://daze.ho.ua>

# Basics to repeat

- Data segment: `.data`
- Code segment: `.text`
- Label (`main:`) – address of memory
- Entry point: `.global main`
- Move data: `mov`
- Length suffix: `b`, `w`, `l`, `q`
- Arithmetic: `add`, `sub`, `imul`, `idiv`, ...
- Registers: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rbp`, `rsp`, ...

# Moving data from memory to register

.data

num: .long -88

.text

...

**movl num, %eax**                   # from num to %eax

- move 4 bytes (32 bits) from memory location addressed by num to register %eax

# Moving data between registers

**`movl %eax, %ebx`**      # from %eax to %ebx

- move 4 bytes (32 bits) from register %eax to register %ebx

# Moving data from register to memory

.data

num: .long 0

.text

...

**movl %eax, num**                   # from %eax to num

- move 4 bytes (32 bits) from register %eax to memory location addressed by num

# Arithmetic

- #  $c = (((a+b)*b)-b)/b$ , say  $a=200$ ,  $b=-300$

movl a, %ecx                   # %ecx=a

movl b, %ebx                   # %ebx=b

addl %ebx, %ecx               # %ecx=a+b, %ecx=-100

imul %ebx, %ecx               # %ecx=(a+b)\*b, %ecx=30000

subl %ebx, %ecx               # %ecx=(a+b)\*b-b, %ecx=30300

movl %ecx, %eax

cdq                           # extend operand: cbw, cwd

idiv %ebx                   # %eax=(((a+b)\*b)-b)/b, %eax=-101

movl %eax, c

# Basic logic (bitwise)

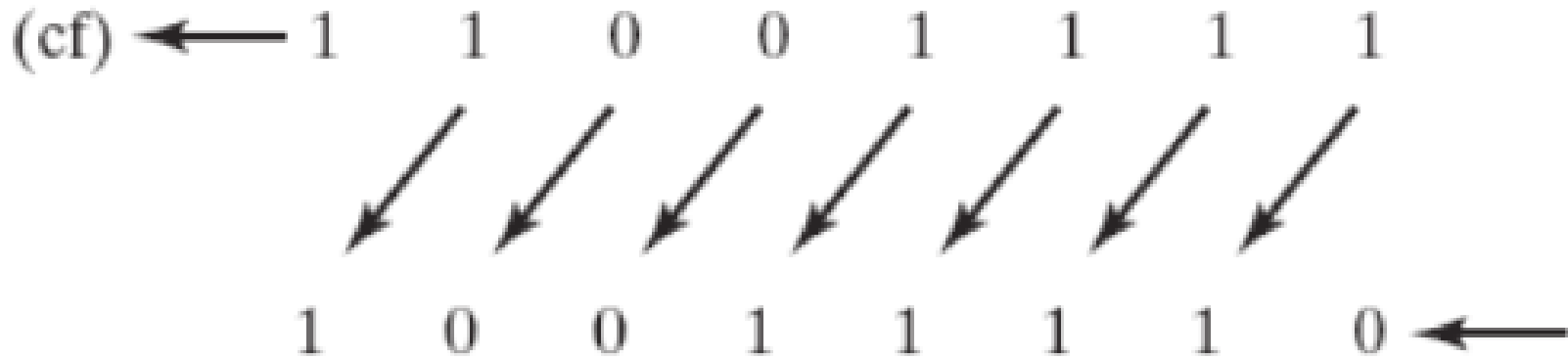
- conjunction (logical and)  
    **and** mask, destination
- disjunction (logical)  
    **or** addend, destination
- exclusive or  
    **xor** flip, destination
- logical negation  
    **not** argument

# Logical Shift Instructions

- right shift  
**shr** cnt, dest
- left shift  
**shl** cnt, dest
- the bits that slide off the end disappear, except for the last, which goes into the carry flag, and the spaces are filled with zeros



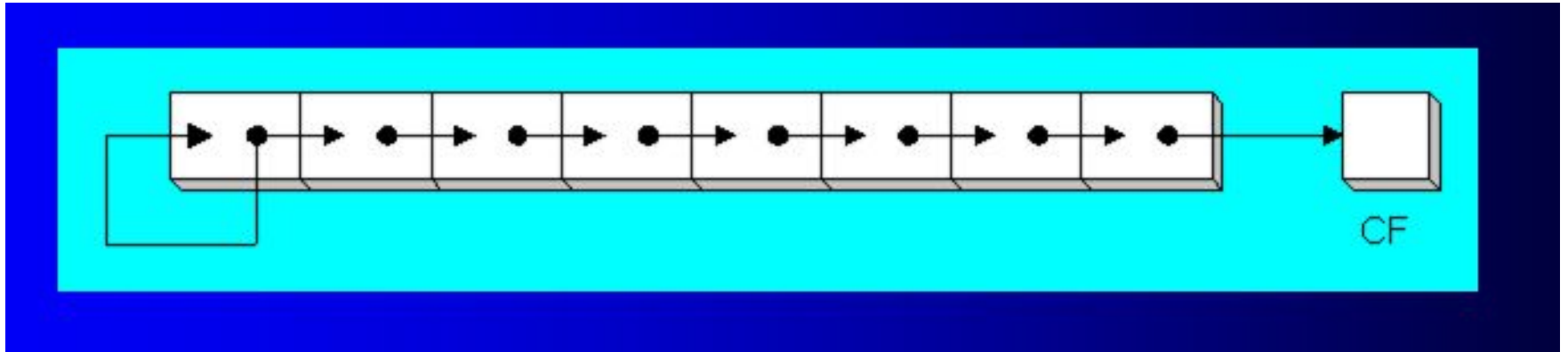
# Logical Shift Instructions scheme



# Arithmetic Shift Instructions

- right shift  
    **sar** cnt, dest
- left shift  
    **sal** cnt, dest
- the bits that slide off the end disappear, except for the last, which goes into the carry flag, but in an arithmetic shift, the spaces are filled in such a way to preserve the sign

# Arithmetic Shift Instructions scheme

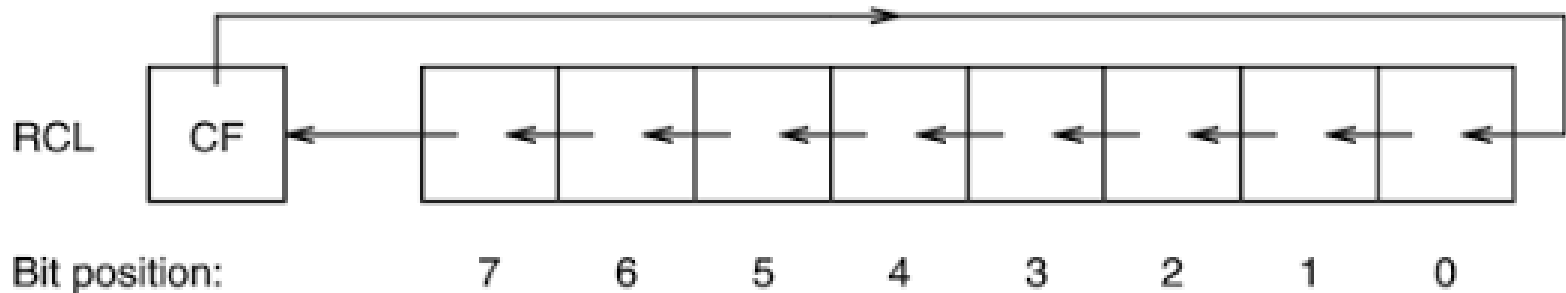
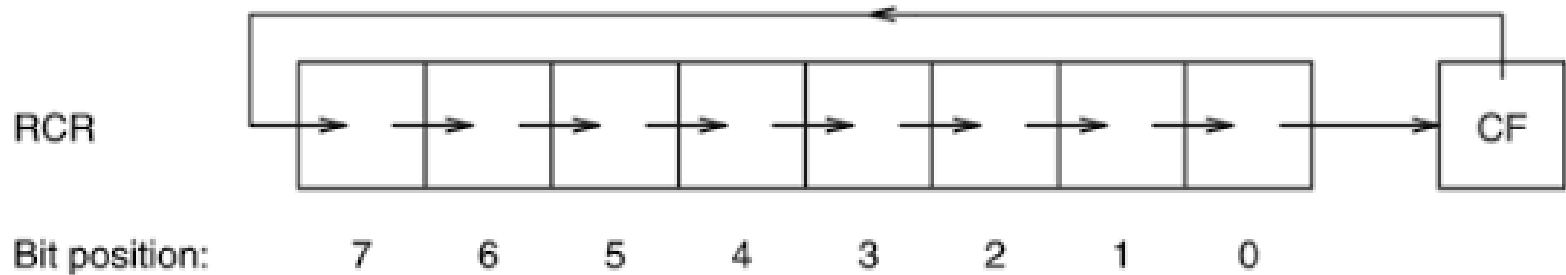


**sar**

# Rotate Instructions

- rotate right
  - ror** offset, variable
  - rcr** cnt, dest # with carry bit
- rotate left shift
  - rol** offset, variable
  - rcl** offset, variable # with carry bit

# Rotate Instructions scheme



# Task

- For a given IP address, number of bits of network address, and a new host address
- Obtain:
  - network address
  - broadcasting address
  - host address replaced

# Jump instructions

- Assign specified value to EIP
- Unconditional Jump  
**jmp** loc
- EIP=loc
- The location passed as the argument is usually a label. The first instruction executed after the jump is the instruction immediately following the label.

# Conditional jump instructions – compare then jump

- **je** loc      # Jump if Equal
- **jne** loc    # Jump if Not Equal
- **jg** loc      # Jump if Greater
- **jge** loc    # Jump if Greater or Equal
- **jl** loc      # Jump if Lesser
- **jle** loc    # Jump if Less or Equal
- ...



# Branching pattern

- **if** condition **then** operands1 **else** operands2

**cmp** source destination # condition s=d

**jne** else

then: <operands1>

**jmp** goon

else: <operands2>

goon: ...

# Test odd or even

|       |                |     |                 |
|-------|----------------|-----|-----------------|
|       | .data          |     | test %eax, %eax |
| x:    | .long 11       |     | jnz le          |
| o:    | .asciz "odd"   |     | mov \$e, %rdi   |
| e:    | .asciz "even"  |     | jmp p           |
|       | .text          | le: | mov \$o, %rdi   |
|       | .global main   | p:  | call puts       |
| main: | movl x, %eax   |     | xor %rax, %rax  |
|       | andl \$1, %eax |     | ret             |

# Loop pattern

- **while** condition **do** operands **done**

```
loop: cmp source destination # condition s=d  
      jne goon  
      <operands>  
      jmp loop  
goon: ...
```

# Sum of arithmetic sequence

```
.data
a0:   .long 1
da:   .long 3
n:    .long 3
s:    .long 0
f:    .asciz "%d\n"

.text

.global main

main: movl a0, %ebx
      movl %ebx, %eax
      movl da, %edx
      movl n, %ecx
```

```
l:    test %ecx, %ecx
      jz go
      addl %edx, %ebx
      addl %ebx, %eax
      decl %ecx
      jmp l

go:   movl %eax, %s
      mov $f, %rdi
      movl %s, %esi
      xor %rax, %rax
      call printf
      xor %rax, %rax
      ret
```

# Addressing modes

- Register – operand in specified register (%)
- Direct memory – operand in specified (by a label) location of memory
- Immediate – operand inside instruction (\$)
- **Direct-Offset Addressing**
- **Indirect Memory Addressing**

# Direct-Offset Addressing example

```
        .data
a:      .long 1,2,3,4,5
c:      .long 0
format: .asciz "%d\n"
        .text
        .global main
main:   movl $0, %eax
        addl a, %eax
        addl a+4, %eax
        addl a+8, %eax
```

```
        addl a+12, %eax
        addl a+16, %eax
        movl %eax, c
        mov $format, %rdi
        movl c, %esi
        xor %rax, %rax
        call printf
        xor %rax, %rax
        ret
```

# Indirect Memory Addressing

- This addressing mode utilizes the computer's ability of Segment:Offset addressing. Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.
- Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register.

# Indirect Memory Addressing example 1

```
        .data
a:      .long 1,2,3,4,5
c:      .long 0
format:.asciz "%d\n"
        .text
        .global main
main:    movl $0, %eax
        mov $a, %rbx
        addl (%rbx), %eax
        addl 4(%rbx), %eax
```

```
        addl 8(%rbx), %eax
        addl 12(%rbx), %eax
        addl 16(%rbx), %eax
        movl %eax, c
        mov $format, %rdi
        movl c, %esi
        xor %rax, %rax
        call printf
        xor %rax, %rax
        ret
```



# Indirect Memory Addressing example 2

```
        .data
a:      .long 1,2,3,4,5
c:      .long 0
format:.asciz "%d\n"
        .text
        .global main
main:   movl $0, %eax
        mov $a, %rbx
        addl (%rbx), %eax
        add $4, %rbx
        addl (%rbx), %eax
```

```
add $4, %rbx
addl (%rbx), %eax
add $4, %rbx
addl (%rbx), %eax
movl %eax, c
mov $format, %rdi
movl c, %esi
xor %rax, %rax
call printf
xor %rax, %rax
ret
```