

# Bit Operations

ICS312 - Spring 2009  
Machine-Level and  
Systems Programming

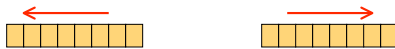
Henri Casanova (henric@hawaii.edu)

## Why bit operations

- Assembly languages all provide ways to manipulate bits
- Some of the coolest “tricks” in assembly rely on bit operations
  - Only a few instructions can do a lot very quickly using judicious bit operations
- Let’s look at some of the common operations, starting with **shifts**
  - logical shifts
  - arithmetic shifts
  - rotate shifts

## Shift Operations

- A shift moves the bits around in some data
- A shift can be toward the left (i.e., toward the most significant bits), or toward the right (i.e., toward the least significant bits)



- There are two kinds of shifts:
  - Logical Shifts
  - Arithmetic Shifts

## Logical Shifts

- The simplest shifts: bits disappear at one end and **zeros appear at the other**

original byte	1 0 1 1 0 1 0 1
left log. shift	0 1 1 0 1 0 1 0
left log. shift	1 1 0 1 0 1 0 0
left log. shift	1 0 1 0 1 0 0 0
right log. shift	0 1 0 1 0 1 0 0
right log. shift	0 0 1 0 1 0 1 0
right log. shift	0 0 0 1 0 1 0 1

## Logical Shift Instructions

- Two instructions: **shl** and **shr**
- For each you can specify by how many bits you want to do the shift
  - Either by just passing a constant to the instruction
  - Or by using whatever is stored in the CL register
- After the instruction executes, the carry flag (CF) contains the (last) bit that was shifted out
- Example:

```
mov     al, 0C6h      ; al = 1100 0110
shl     al, 1         ; al = 1000 1100 (8Ch) CF=1
shr     al, 1         ; al = 0100 0110 (46h) CF=0
shl     al, 3         ; al = 0011 0000 (30h) CF=0
mov     cl, 2
shr     al, cl        ; al = 0000 1100 (0Ch) CF=0
```

## Shifts and Numbers

- The main use for shifts: **quickly** multiply and divide by powers of 2
- In decimal
  - multiplying 0013 by 10 amounts to doing one left shift to 0130
  - multiplying by 100 amounts to doing two left shifts to 1300
- In binary
  - multiplying by 00101 by 2 amounts to doing a left shift to 01010
  - multiplying by 4 amounts to doing two left shifts to 10100
- If numbers are too large, then we’d need more bits and multiplication doesn’t produce valid results
  - e.g., 10000000 (128d) cannot be left-shifted to obtain 256 using 8-bit values
- Similarly, dividing by powers of two amounts to doing right shifts:
  - right shifting 10010 (18d) leads to 01001 (9d)
- Note that when dividing odd numbers by two we “lose bits”, which amounts to rounding to the lower integer quotient
  - Consider number 10011 (19d)
  - Right shift: 01001 (9d - rounded below)
  - Right shift: 00100 (4d - rounded below)

## Shifts and Unsigned Numbers

- Using the shifts works only for unsigned numbers
- When numbers are signed, the shifts do not handle the sign bits correctly and cannot be interpreted as multiplying/dividing by powers of 2 anymore
- Example: Consider the 1-byte number FE
  - If Unsigned:
    - FE = 254d = 11111110b
    - right shift: 01111111b = 7Fh = 127d (which is 254/2)
  - In Signed:
    - FE = -2d = 11111110b
    - right shift: 01111111b = 7Fh = +127d (which is NOT -2/2)

## Arithmetic Shift Example

- If **signed numbers**, then the operations below are correct multiplications / divisions of 1-byte quantities

```
mov  al, 0C3h      ; al = 1100 0011 (-61d)
sal  al, 1         ; al = 1000 0110 (86h = -122d)
sar  al, 3         ; al = 1111 0000 (F0h = -16d)
                        ; (note that this is not an exact division as we
                        ; lose bits on the right!)
```
- The following is not a correct multiplication by 16!

```
sal  al, 4         ; al = 0000 0000 (0d, which can't be right)
```
- One should use the `imul` instruction instead (but unfortunately `imul` doesn't work on 1-byte quantities):

```
movsx ax, al      ; sign extension
imul  ax, 16      ; result in ax
```
- Let's see an example in file `ics312_arithmetic_shift.asm`

## In-Class Exercise

- Consider the following instructions

```
mov  ax, 0F471h
      F471      CF=0
sar  ax, 3
      FE8E      CF=0
shl  ax, 7
      4700      CF=1
sar  ax, 10
      0011      CF=1
```

## Arithmetic Shifts

- Since the logical shifts do not work for signed numbers, we have another kind of shifts called arithmetic shifts
- Left shift: **sal**
  - This instruction works just like `shl`
  - As long as the sign bit is not changed by the shift, the result will be correct (i.e., will be multiplied by 2)
- Right shift: **sar**
  - This instruction does NOT shift the sign bit: the new bits entering on the left are copies of the sign bit
- Both shifts store the last bit out in the carry flag

## In-Class Exercise

- Consider the following instructions

```
mov  ax, 0F471h
sar  ax, 3
shl  ax, 7
sar  ax, 10
```
- At each step give the content of register `ax` (in hex) and the value of `CF`

## Rotate Shifts

- There are more esoteric shift instructions
- **rol** and **ror**: circular left and right shifts
  - bits shifted out on one end are shifted in the other end
- **rcl** and **rcr**: carry flag rotates
  - the source (e.g., a 16-bit register) and the carry flag are rotated as one quantity (e.g., as a 17-bit quantity)
- See the book (Section 3.1.4) for more detailed descriptions and examples

## Example Using Shifts

- Let's go through the Example 3.1.5 in the book
- Say you want to count the number of bits that are equal to 1 in register EAX
- One easy way to do this is to use shifts
  - Shift 32 times
  - Each time the carry flag contains the last shifted bit
  - If the carry flag is 1, then increment a counter, otherwise do not increment a counter
  - When you're done the counter contains the number of 1's
- Let's write this in x86 assembly
  - The textbook has it written a bit differently (uses the loop instruction)

## Example Using Shifts

```
; Counting 1 bits in EAX
mov  bl, 0      ; bl is the number of 1 bits
mov  cl, 32     ; cl is the loop counter
loop_start:
    shl  eax, 1  ; left shift
    jnc  not_one ; if carry != 1, jump to not_one
    inc  bl      ; increment the number of 1 bits
not_one:
    dec  cl      ; decrement the loop counter
    jnz  loop_start ; if more iterations goto loop_start
```

## The same, with the adc instruction

- Convenient instruction: **adc** (add carry)
  - `adc dest, src ; dest += src + cf`

```
; Counting 1 bits in EAX
mov  bl, 0      ; bl is the number of 1 bits
mov  cl, 32     ; cl is the loop counter
loop_start:
    shl  eax, 1  ; left shift
    adc  bl, 0   ; add the carry to bl
    dec  cl      ; decrement the loop counter
    jnz  loop_start ; if more iterations goto loop_start
```

## The same, with the loop instruction

- Remember the **loop** instruction
  - `loop <label> ; decrements loop index (in ecx) ; and branches if ecx isn't 0`

```
; Counting 1 bits in EAX
mov  bl, 0      ; bl is the number of 1 bits
mov  ecx, 32    ; ecx is the loop counter
loop_start:
    shl  eax, 1  ; left shift
    adc  bl, 0   ; add the carry to bl
    loop loop_start ; decrement ecx and loop if needed
```

## Boolean Bitwise Operations

- There are assembly bitwise instructions for all standard boolean operations: AND, OR, XOR, and NOT
- Bits are computed individually
- Examples:

AND	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	1	0	1	1	0	0	1	1	0	1	1	0	1	0	0	1	0	0	OR	<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	1	1	0	0	0	1	0	1	1	0	1	1	1	1	1	0	1	1
1	0	1	1	0	0																																		
1	1	0	1	1	0																																		
1	0	0	1	0	0																																		
1	1	0	0	0	1																																		
0	1	1	0	1	1																																		
1	1	1	0	1	1																																		
=		=																																					
XOR	<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	1	1	0	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0	NOT	<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	0	0	0	1	0	0	1	1	1	0						
1	1	0	0	0	1																																		
0	1	1	0	1	1																																		
1	0	1	0	1	0																																		
1	1	0	0	0	1																																		
0	0	1	1	1	0																																		
=		=																																					

## Boolean Bitwise Instructions

```
mov  ax, 0C123h

and  ax, 82F6h ; ax = C123 AND 82F6 = 8022

or   ax, E34Fh ; ax = 8022 OR E34F = E36F

xor  ax, 36E9h ; ax = E36F XOR 36E9 = D586

not  ax ; ax = NOT D586 = 2A79
```

## The test Instruction

- The **test** instruction performs an AND, but does not store the result
- It only sets the FLAG bits
  - Just like **cmp** does a subtraction but never stores its result
- Note that all boolean bitwise instructions do set the FLAG bits, **BUT** for the **not** operation, which doesn't
- Example:

```

mov  al, 0FFh          mov  al, 0FFh
test al, 00h          not  al
jz   foo ; branches   jz   foo ; does not branch
    
```

## Uses of Bitwise operations

- Bitwise operations are very useful to modify individual bits within data
- This is done via "**bit masks**", that is constant (immediate) quantities with carefully chosen bits
- Example:
  - Say you want to turn on bit 3 of a 2-byte value (counting from the right, with bit 0 being the least significant bit)
  - An easy way to do this is to OR the value with 0000000000001000, which is 8 in decimal
  - Say the value is stored in **ax**
  - You can simply execute the command:
 

```
or  ax, 8 ; turns on bit 3 in ax
```
- Easy to generalize
  - To turn on bits: use OR (with appropriate 1's in the bit mask)
  - To turn off bits: use AND (with appropriate 0's in the bit mask)
  - To flip bits: use XOR (with appropriate 1's in the bit mask)

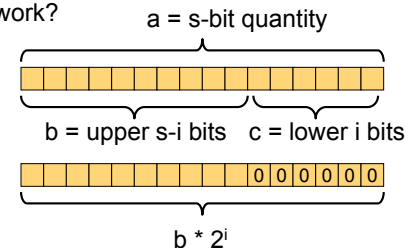
## Bit Mask Operations Examples

```

mov  eax, 04F346BA2h
or  ax, 0F000h ; turns on 4 leftmost bits of ax
                ; eax = 4F34FBA2
xor  eax, 000400000h ; inverts bit 22 of EAX
                ; eax = 4F74FBA2
xor  ax, 0FFFFh ; 1's complement of ax
                ; eax = 4F74045D
    
```

## Remainder of a Division by $2^i$

- To find the remainder of a division of an operand by  $2^i$ , just **AND** the operand by  $2^i-1$
- Why does this work?



Therefore,  $a = b * 2^i + c$ , an  $c$  is the remainder!  
The remainder is simply the lowest  $i$  bits!

## Remainder of a Division by $2^i$

- Let's compute the remainder of the integer division of 123d by  $2^5=32$ d (unsigned) by doing an AND with  $2^5-1$

```

mov  ax, 123      00000000000011111011
mov  bx, 0001Fh  000000000000000011111
and  bx, ax      0000000000000000111011
    
```

- The remainder when dividing 123 by 32 is 11011b = 27d

## Turning on a specific bit

- Say you want to turn on a specific bit in some data, but that you don't know which one before you run the program
  - the index of the bit to turn on is contained in a register
  - we need to build the bit mask "on the fly"
- Assuming that the index of the bit is initially in **bl**, and that we which to turn on a bit in **eax**

```

mov  cl, bl ; must have the bit index in cl
mov  ebx, 1 ; create a number 0...01
shl  ebx, cl ; shift it left cl times
or   eax, ebx ; turn on the desired bit using
                ; ebx as a mask!
    
```

## Turning off a specific bit

- Turning a bit off requires one more instruction, to generate a bit mask that looks like 1...101..1
- Assuming that the index of the bit is initially in `bl`, and that we wish to turn on a bit in `eax`

```
mov cl, bl ; must have the bit index in cl
mov ebx, 1 ; create a number 0...01
shl ebx, cl ; shift it left cl times
not ebx; ; take the complement!
and eax, ebx ; turn off the desired bit using
; ebx as a mask!
```

## An odd xor

- One often sees the following instruction:  
`xor eax, eax ; eax = 0`
- This is a simple way to set `eax` to 0
- It is useful because its machine code is smaller than that of, for instance, `mov eax, 0`
- Therefore one saves a few bits in the text segment and while the program runs a few bits less will be needed to be loaded from memory, saving perhaps a few cycles
- **Lesson:** One could do everything with operations that look like those of high-level languages, but the good assembly programmer (and the good compiler) will use bit operations to save memory and/or time
- Let's go through the example in Section 3.3, which is a good example of bit operation craziness

## Avoiding Conditional Branches

- Section 3.3 is all about a trick to avoid conditional branches
- You'll see in ICS431 that conditional branches greatly reduce the speed of processors
  - Essentially, one key to making processors go fast is to allow them to know what's coming up next
  - With conditional branches, the processor doesn't know in advance whether the branch will be taken or not
- In many cases, one cannot avoid using conditional branches
  - It's just in the nature of the computation
  - For instance, for a loop
- But in some cases it's possible
- Let's just look at an example that illustrates some of the coolness of bitwise operations

## SETxx Instructions

- The x86 assembly provides a set of instructions that set the value of a byte register (e.g., `al`) or of a byte memory location to **0 or 1** based on a flag
- Set you want to set `al` to 0 if `bx > cx` or to 1 otherwise (all signed)
- With the `setg` instruction you can save a conditional branch:  
; without `setg` ; with `setg`  
`mov al, 1 ; al = 1` ; `cmp bx, cx`  
`cmp bx, cx ; compare` ; `setg al, 0`  
`jng next ; jump if bx <= cx`  
`mov al, 0 ; al = 0`  
`next:`
- Similar instructions: `setz`, `setng`, `sete`, etc.

## Example: max(a,b)

- Say we want to store into `ecx` the maximum of two (signed) numbers, one stored in `eax` and the other one in `[num]`
- Here is a simple code to do this

```
cmp eax, [num]
jge next ; conditional branch
mov ecx, [num]
jmp end
next:
mov ecx, eax
end;
```
- Let's rewrite this without a conditional branch!

## Example: max(a,b)

- To avoid the conditional branch, one needs a SETxx instruction and clever bit masks
- We use a helper register, `ebx`, which we set to all zeros

```
xor ebx, ebx
```
- We compare the two numbers

```
cmp eax, [num]
```
- We set the value of `bl` to 0 or 1 depending on the result of the comparison

```
setg bl
```

  - If `eax > [num]`, `ebx = 1 = 0...01b`
  - If `eax <= [num]`, `ebx = 0 = 0...00b`
- We negate `ebx` (i.e., take 1's complement and add 1)

```
neg ebx
```

  - If `eax > [num]`, `ebx = FFFFFFFFh`
  - If `eax <= [num]`, `ebx = 00000000h`
- This forms the basis for our bitmask

## Example: max(a,b)

- We now have:
  - eax contains one number, [num] contains the other
  - If  $eax > [num]$ ,  $ebx = \text{FFFFFFFFh}$  (we want to "return" eax)
  - If  $eax \leq [num]$ ,  $ebx = \text{000000000h}$  (we want to "return" [num])
- If eax is the maximum and we AND eax and ebx, we get eax, otherwise we get zero
- If [num] is the maximum and we AND [num] and NOT(ebx), we get [num], otherwise we get zero
- So if we compute  $((eax \text{ AND } ebx) \text{ OR } ([num] \text{ AND } \text{NOT}(ebx)))$  we get the maximum!
  - If eax is the maximum ( $ebx = \text{FFFFFFFFh}$ ):
    - $((eax \text{ AND } ebx) \text{ OR } ([num] \text{ AND } \text{NOT}(ebx))) = \text{eax OR } 0 \dots 0 = \text{eax}$
  - If [num] is the maximum ( $ebx = \text{00000000h}$ ):
    - $((eax \text{ AND } ebx) \text{ OR } ([num] \text{ AND } \text{NOT}(ebx))) = 0 \dots 0 \text{ OR } [num] = [num]$
- Let's just write the code to compute  $((eax \text{ AND } ebx) \text{ OR } ([num] \text{ AND } \text{NOT}(ebx)))$

## Example: max(a,b)

- Computing  $((eax \text{ AND } ebx) \text{ OR } ([num] \text{ AND } \text{NOT}(ebx)))$ :

```
mov     ecx, ebx           ; ecx = ebx
and     ecx, eax           ; ecx = eax AND ebx
not     ebx                ; ebx = NOT(ebx)
and     ebx, [num]        ; ebx = [num] AND NOT(ebx)
or      ecx, ebx           ; voila!
```
- Whole program:

```
xor     ebx, ebx           ; ebx = 0
cmp     eax, [num]        ; compare eax and [num]
setg   bl                 ; bl = 1 if eax > [num], 0 otherwise
neg    ebx                ; take one's complement + 1
mov     ecx, ebx           ;
and     ecx, eax           ; ecx = eax AND ebx
not    ebx                ;
and     ebx, [num]        ; ebx = [num] AND NOT(ebx)
or      ecx, ebx           ; voila!
```

## Bit Operations in C

- Although in this course we focus on assembler, let's discuss C a little bit
- C is well-known for allowing the programmer to write code that is either high-level or that looks pretty close to assembly
  - Tries to allow "easy" programming as well as "performance" programming
- One area in which C is most like assembly is in its ability to operate on bits
- This is very useful, and since you probably won't see it too much in other courses, let's go through it
  - Especially because bit operations are used/needed by several important system calls

## Bitwise Operators in C

- Boolean Operations:
  - AND: &
  - OR: ||
  - XOR: ^
  - NOT: !
- Bitwise Operations:
  - AND: &
  - OR: |
  - XOR: ^
  - NOT: ~
- Shift Operations:
  - Left Shift: <<
  - Right Shift: >>
  - Logical if operand is unsigned
  - Arithmetic if operand is signed

## Example Operations

```
short int      s; // 2-byte signed
short unsigned int u; // 2-byte unsigned
s = -1; // s = 0xFFFF
u = 100; // u = 0x0064
u = u | 0x0100; // u = 0x0164
s = s & 0xFFFF0; // s = 0xFFFF0
s = s ^ u; // s = 0xFE94
u = u << 3; // u = 0x0B20
s = s >> 2; // s = 0xFFA5
```

## Common Uses of Bit Operations

- C can use bit operations like assembly
  - Typically for fast multiplications, divisions
- The most common use is for dealing with file permissions
- The POSIX API, used to deal with files on all Linux systems, uses bits to encode file access permissions
- If you have to write a C code that needs to read/modify file permissions, then you need to use C's bit operations

## Using chmod from C

- In a POSIX system, there is a C library function called `chmod()` that modifies permissions
- `chmod()` takes two arguments:
  - The file name
  - A 4-byte quantity that is interpreted as a bunch of individual bits, which describe the permission
- To make life easy, the user does not have to construct the bits by hand, but there are macros
- For instance:

```
chmod("file", S_IRUSR)
```

Gives read permission to the owner of the file  
`S_IRUSR` has one of its bits turned on
- This makes it easy to do multiple things at once:

```
chmod("file", S_IRUSR | S_IWUSR | S_IROTH)
```

The user can read and write, all "other" users can read
- Simply use a bitwise or to apply all permission settings

## Counting Bits

- Section 3.6 of the book shows many methods for counting bits
- These methods are shown in C, but of course it's easy (if perhaps cumbersome) to implement them in assembly
- Let's look at Method #1
  - Make sure you look at the others on your own and that you understand them (some are quite creative)

```
unsigned char data; // 1 byte (book uses 4)
char count; // counter (only 1 byte necessary)
while (data) {
    data = data & (data - 1);
    cnt++;
}
printf("number of 1 bits: %d\n", cnt);
```

## In-class Quiz

- Quiz #5 will be on this set of lecture notes
  - Basic Assembly Language II
  - Bit operations
- The quiz will be on...

## Modifying Permissions

- Say you want to write a program that, given a file, removes write access to others and adds read access to the owner of the file
- First step: get the 4-byte permission data

```
struct stat s; // data structure
stat("file", &s);
unsigned int p; // 4-byte quantity
p = s.st_mode;
```
- Second step: modify it, keeping most bits as they were

```
chmod("file", (p & ~S_IWOTH) | S_IRUSR);
```

## Counting Bits

```
while (data) {
    data = data & (data - 1);
    cnt++;
}
```

- Example: `data = 01011010` (in binary)
  - `data = data & (data - 1) = 01011010 & 01011001 = 01011000`
  - `data = data & (data - 1) = 01011000 & 01010111 = 01010000`
  - etc.
- At each step, we set the rightmost 1 bit to 0!
- When we have all zeros we stop
- The number of iterations is the number of 1 bits