Bitmasks

ICS312
Machine-Level and
Systems Programming

Henri Casanova (henric@hawaii.edu)

**Boolean Bitwise Operations**

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

  &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&n
Uses of Bitwise operations

- Bitwise operations are useful to modify individual bits within data
- This is done via bit masks, i.e., 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 instruction:
    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

```assembly
mov eax, 04F36BA2h
or ax, 0F000h  ; turns on 4 leftmost bits of ax
                ; eax = 4F34FBA2
xor eax, 00400000h ; 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?
  - a = s-bit quantity
  - b = upper s-i bits
  - c = lower i bits
  - b * 2^i

Therefore, a = b * 2^i + c, an c is the remainder!
The remainder is simply the lowest i bits!
**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 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
  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 on 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**: On 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
- 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/craziness 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:

```assembly
; without setg
mov al, 1 ; al = 1
cmp bx, cx
jng next ; jump if bx <= cx
mov al, 0 ; al = 0
```

```assembly
; with setg
mov al, 1
setg al, 0
```

- 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:

```assembly
cmp eax, [num] ; conditional branch
jge next
mov ecx, [num]
jmp end
next:
mov ecx, eax
```

- Let’s rewrite this without a conditional branch!
  - Conditional branches are bad for performance

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:

```assembly
xor ebx, ebx
```

- We compare the two numbers:

```assembly
cmp eax, [num]
```

- We set the value of bl to 0 or 1 depending on the result of the comparison:

```assembly
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):

```assembly
neg ebx
```

- If eax > [num], ebx = FFFFFFFFh
- If eax <= [num], ebx = 0000000000h

- We now have:
  - eax contains one number, [num] contains the other
  - If eax > [num], ebx = FFFFFFFFh (we want to “return” eax)
  - If eax <= [num], ebx = 0000000000h (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 AND ebx) OR ([num] AND NOT(ebx))) we get the maximum!
  - If eax is the maximum (ebx = FFFFFFFFh):
    - ((eax AND ebx) OR ([num] AND NOT(ebx))) = eax OR 0...0 = eax
  - If [num] is the maximum (ebx = 0000000000h):
    - ((eax AND ebx) OR ([num] AND NOT(ebx))) = 0...0 OR [num] = [num]

- Let’s just write the code to compute ((eax AND ebx) OR ([num] AND NOT(ebx)))
Example: max(a,b)

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

- Whole program:
  
  ```assembly
  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: XXX
  - 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

```c
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 = 0xFFF
```
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: (p contains the file’s permission bits)
  chmod("file", p | 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", p | 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

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); // get all file metadata
  unsigned int p; // 4-byte quantity
  p = s.st_mode; // p = permission bits
- Second step: modify, keeping most bits unchanged
  chmod("file", (p & ~S_IWOTH) | S_IRUSR);

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",count);
Counting Bits

```cpp
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