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 (i.e., “bitwise”)
- Examples:

AND

	1	0	1	1	0	0	0	0
AND	1	1	0	1	1	0	1	1
=	1	0	0	1	0	0	0	0

OR

	1	1	0	0	0	1	0	1
OR	0	1	1	0	1	1	1	0
=	1	1	1	0	1	1	1	1

XOR

	1	1	0	0	0	1	0	1
XOR	0	1	1	0	1	1	1	0
=	1	0	1	0	1	0	1	1

NOT

NOT	1	1	0	0	0	1	0	1
=	0	0	1	1	1	0	1	0

Boolean Bitwise Instructions

```
mov   ax, 0C123h
and   ax, 082F6h  ; ax = C123 AND 82F6    = 8022
or    ax, 0E34Fh  ; ax = 8022 OR E34F     = E36F
xor   ax, 036E9h  ; 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
test    al, 000h
jz      foo       ; branches
```

```
mov     al, 0FFh
not     al
jz      foo  ; may not branch
```

Uses of Bitwise operations

- Bitwise operations are used to modify particular bits in data
- This is done via bit masks, i.e., constant (or “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
```

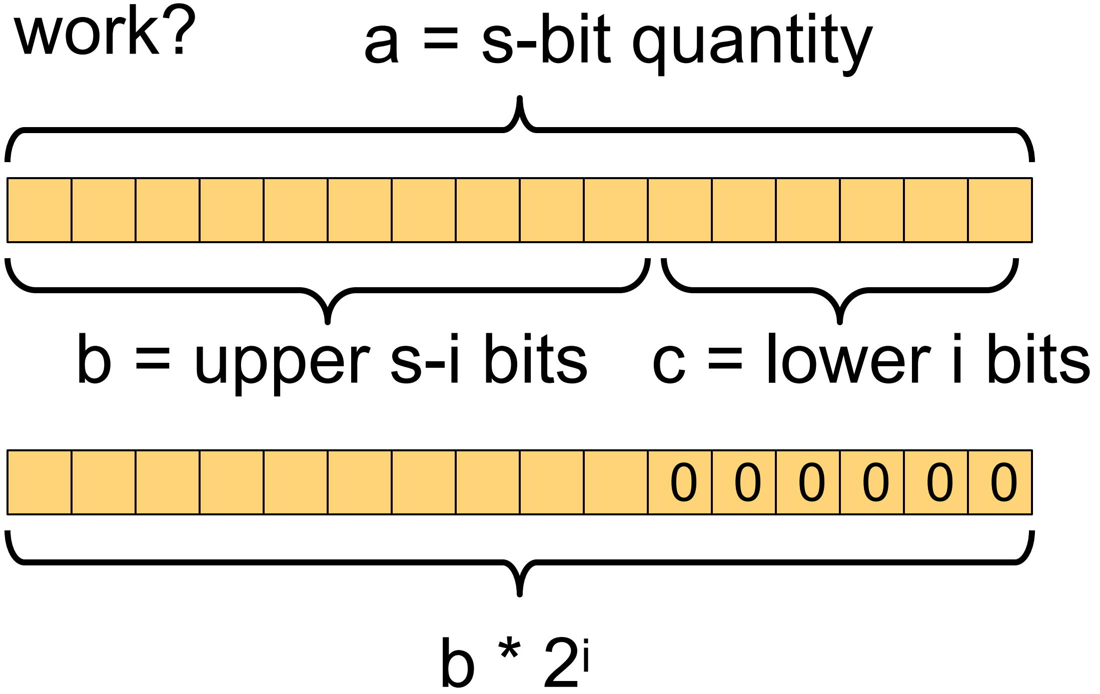
- Easy to generalize
  - To turn on bits: use OR (with appropriate 1’s in the bit mask)
  - To turn off bits: use AND (with appropriate 0’s in the bit mask)
  - To flip bits: use XOR (with appropriate 1’s in the bit mask)

Bit Mask Operations Examples

mov eax, 04F346BA2h	
or ax, 0F000h	; turns on 4 leftmost bits of ax
	; eax = 4F34FBA2
xor eax, 000400000h	; flips bit 22 of EAX
	; eax = 4F74FBA2
xor ax, 0FFFFh	; 1’s complement of ax
	; eax = 4F74045D
	; (same as doing “not”)

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!

Another Way to Look at it

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

=

x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0

+

0	0	0	0	0	0	0	0	0	0	0	x	x	x	x	x

Example with i=5

Another Way to Look at it

a	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
								=								
	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0
								+								
	0	0	0	0	0	0	0	0	0	0	0	x	x	x	x	x

Multiple of $2^5 = 32$

Smaller than 2^5=32

Therefore, a = <some multiple of 32> +
<some number smaller than 32>

By definition, that smaller than 32 number is the remainder of the division of a by 32

It is just the low $\log_2(32)$=5 bits of a!!!

Remainder of a Division by 2^i

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

```
mov  ax, 123
mov  bx, 0001Fh
and  bx, ax
```

0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	ax
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	bx
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	bx

- The remainder when dividing 123 by 32 is 11011b = 27d
- Often one says "I masked out the 11 high bits of ax" (using bx as the bitmask)

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 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
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, meaning that when the program runs it will load fewer bits less from memory, saving perhaps a few cycles
- **Message**: One could do everything with high-level operations, but the good (assembly) programmer (and definitely 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 kill performance
  - 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, in a foor loop
- But in some cases it's possible, with some cleverness
- 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
- Say 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
mov   al, 1    ; al = 1
cmp   bx, cx   ; compare
jng   next     ; jump if bx <= cx
mov   al, 0    ; al = 0
next:
```

```
; with setg
cmp    bx, cx
setg   al, 0
```

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
mov   al, 1     ; al = 1
cmp   bx, cx    ; compare
jng   next      ; jump if bx <= cx
mov   al, 0     ; al = 0
next:
```

```
; with setg
cmp     bx, cx
setg    al, 0
```

The second operand is added to al "after" al is set to 0 or 1. In this case we don't want to add anything, so we use "0"

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!
- Just follow along… this won't strike you as intuitive but you should be able to understand that it works
  - Even though you wouldn't have been able to do it from scratch on your own, most likely

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          ; ebx = 0
```

- We compare the two numbers

```
cmp     eax, [num]        ; compare eax to [num]
```

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

```
setg    bl                ; set bl to 0 or 1
```

  - 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 = 000000000h

Example: max(a,b)

- 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 = 00000000h):
    - ((eax AND ebx) OR ([num] AND NOT(ebx))) = 0...0 OR [num] = [num]
- Now we just need to compute ((eax AND ebx) OR ([num] AND NOT(ebx)))

Example: max(a,b)

- Computing ((eax AND ebx) OR ([num] AND NOT(ebx))):

```
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:

```
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!
```

Predicates

- The technique we have used to remove a conditional branch is often called “predicates”
- Essentially, you replace conditional branches with Boolean computations that “work out”
- Predicates are (or should be) used extensively when programming for the GPU
  - For performance reasons
- Whether the previous example leads to performance gains is unclear:
  - Version #1: we have a “bad” branch instruction
  - Version #2: we have a lot of computation
- But if we have a lot of maximum computations to perform in a row, Version #2 is almost certainly faster

Bitwise Operations in High-Level Languages

- Although in this course we focus on assembly programming, let's discuss high-level languages a little bit
- All high-level languages provide bitwise operations
- Let's look at the typical bitwise operators…

Operators in C/C++/Java/…

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

Example Operations: C/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 & 0xFFF0;           // s = 0xFFF0
s = s ^ u;                // s = 0xFE94
u = u << 3;               // u = 0x0B20
s = s >> 2;               // s = 0xFFA5
```

Common Uses of Bit Operations

- One can use bit operations in high-level languages like in assembly
  - e.g., fast multiplications and divisions
- A very common use is for dealing with file permissions on UNIX systems
- The POSIX API, for dealing with files on all Linux systems, uses bits to encode file access permissions
- If you have to write code that needs to read/modify file permissions, then you need to use bitwise 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 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

```
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!!

Detecting duplicates

- Inspired from the first exercise in the “Cracking the Coding Interview” book
- You have write a function that takes an ASCII character as input, and returns true is that function was called previously with that ASCII character
- Typical idea: maintain a set of “already seen” characters
  - e.g., a <char, boolean> map, an array of boolean
- But storing a boolean in a byte is super wasteful
  - You’re wasting 7 out of 8 bits
- The book suggests using a “bit field” to save space
- Let’s do it live… (could be bad ;))

Conclusion

- Next week we'll have an in-class practice quiz about this module

- Let's look at Homework assignment #6