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
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
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
0	1	1	0	1	1	1	0
1	0	1	0	1	0	1	1

=

NOT

1	1	0	0	0	1	0	1
0	0	1	1	1	0	1	0

=

Example

```
mov  al, 10011111b

and al, 11100100b
   ;al = 10000100

or  al, 01101010b
   ;al = 11101110

xor al, 01011011
   ;al = 10110101
```

Uses of Boolean Bitwise operations

- Bitwise operations are used to modify particular bits in data
- This is done via bitmasks, 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 do:
    - `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

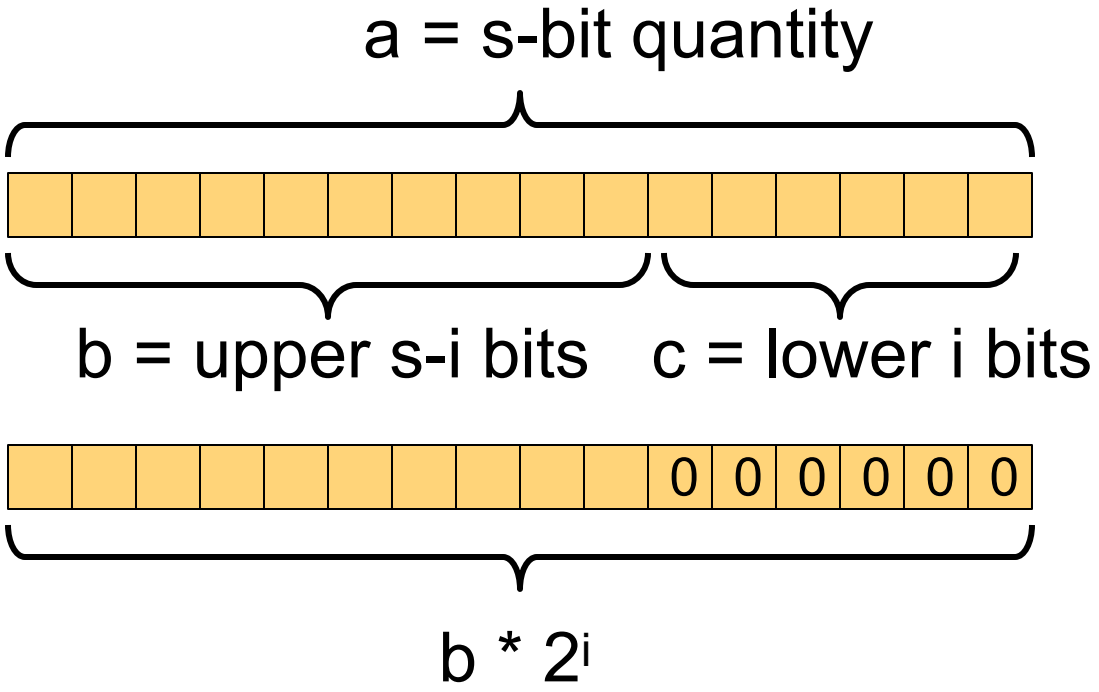
```
mov eax, 04F346BA2h
or ax, 0F000h    ; turns on 4 leftmost
                 ; bits of ax
                 ; eax = 4F34FBA2

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?

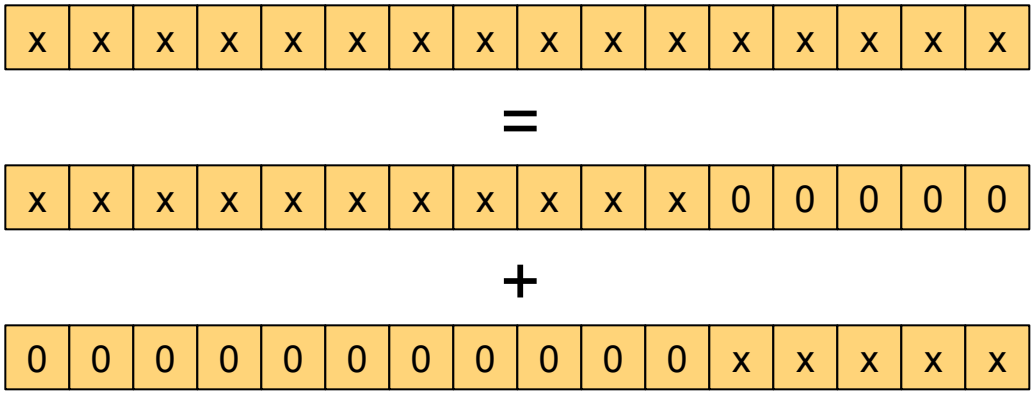


Therefore, $a = b * 2^i + c$, an c is the remainder!

The remainder is simply the lowest i bits!

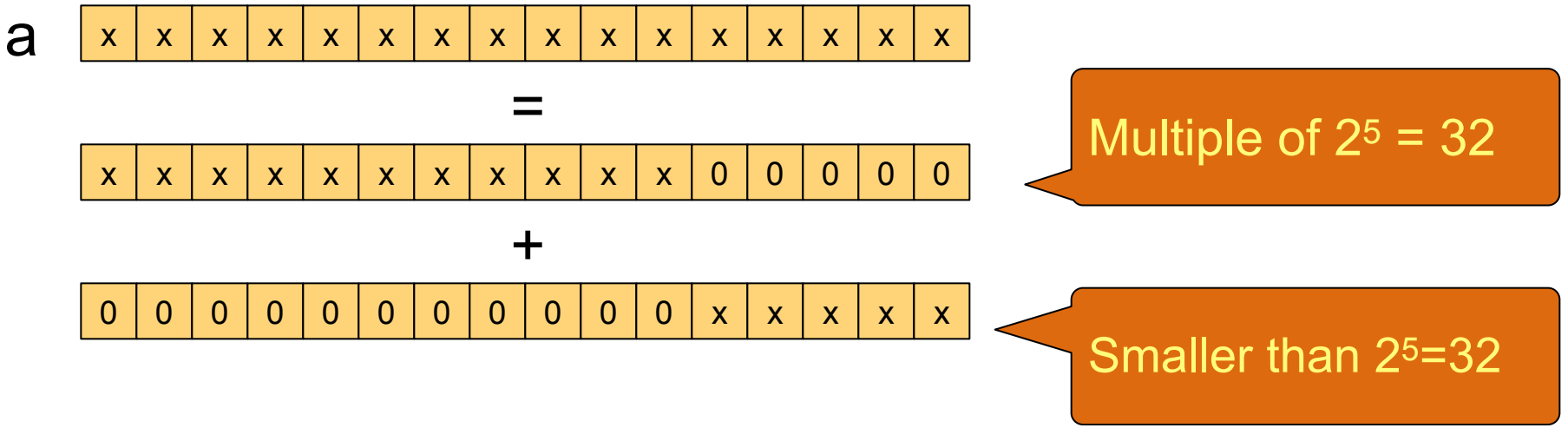
Another Way to Look at it

a



Example with i=5

Another Way to Look at it



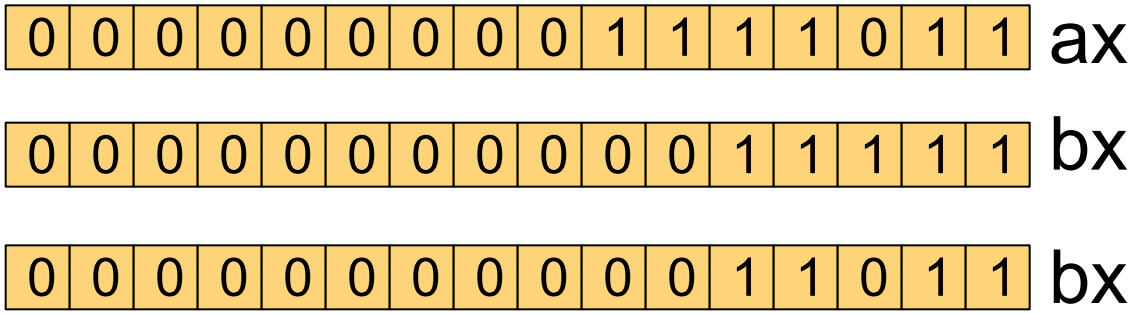
Therefore, a = <some multiple of 32> +
<some number smaller than 32>

By definition, the “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!!!

Example

- 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
```



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

Live-coding (if time)

- Let's write code that prints the hex representation of a 32-bit number…
  - For this we need to use shifts and very simple bitmasks
  - Printing the binary representation is basically what we just did for counting the number of bits set to 1 in the previous set of lecture notes
    - But instead of adding the carry to a counter we print a 0 or 1 depending on whether the carry is 0 or 1
  - This is a bit more involved

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
```

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
- Feel free to go through the example in Section 3.3, which is a good example of bit operation “craziness”

The classic “swap two values”

- Say you have a value in EAX and a value in EBX and you want to swap them **without** using a third register
- Well-known trick:

```
XOR     EAX, EBX
XOR     EBX, EAX
XOR     EAX, EBX
```

- It works because of the fundamental properties of XOR
  - Commutativity: A ^ B = B ^ A
  - Associativity: A ^ (B ^ C) = (A ^ B) ^ C
  - And of course: A ^ A = 0; A ^ 0 = A
- Let’s show that the code above does the right thing…

The classic “swap two values”

```
; Let's use red to denote the original
; values in EAX and EBX

XOR    EAX, EBX
  ; EAX = EAX ^ EBX

XOR    EBX, EAX
  ; EBX = EBX ^ EAX = EBX ^ (EAX ^ EBX)
                     = EAX ^ (EBX ^ EBX)
                     = EAX   ; SWAPPED!

XOR    EAX, EBX
  ; EAX = EAX ^ EBX =(EAX ^ EBX) ^ EAX
                     = EBX ^ (EAX ^ EAX)
                     = EBX   ; SWAPPED!
```

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
- It's called "programming with predicates"
  - Very useful on GPUs
- The example in the book is a bit crazy

Bitwise Operations in High-Level Languages

- All high-level languages provide bitwise operations
- Let's look at the typical bitwise operators…

Operators in C/C++/Java/Python/..

- 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 (0…010…0)

- 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 in High-Level Languages

- We have seen how to use shifts to count bits (set to one) in a value in the previous set of lecture notes
- It was pretty easy by using the carry bit
- But in high-level languages, we don't have access to the carry bit
- Section 3.6 of the textbook shows three methods for counting bits
- Let's look at the first two
  - The third one is a bit complicated, but clever

Method #1

```
unsigned int data;
char count=0;
while (data) {
  data = data & (data -1); // weird???
  count++;
}
printf("number of 1 bits: %d\n",count);
```

- Let's see on an example why this method works..

Method #1

```
unsigned char data; // 1 byte
char count=0;
while (data) {
  data = data & (data -1);
  count++;
}
```

- 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!!

Method #2: More Space, Less Time

- In ICS311 you have learned about time- and space-complexity
- Sometimes, it's a good idea to increase space-complexity to improve time-complexity
- For instance:
  - Precompute a bunch of things, and store them in a table
  - Later lookup that table to "compute" something with low time-complexity, e.g., O(1)
- As long as you need to perform the computation a large number of times, the overhead to precompute the table is negligible
  - And in fact, the table can simply be hardcoded as a bunch of constant values in the code

Method #2: How it works

- The key idea is to precompute the number of bits set to 1 for all possible 1-byte values
  - There are only 256 such values
  - And the “table” can be just an array, indexed by the 1-byte value!
  - And we do the computation using Method #1
- Then, for each each byte in a 4-byte value, one can “compute” the number of bits with one lookup
  - This doesn’t change the asymptotic computational complexity, but reduces the complexity by a constant factor 8x (or close)
- Let’s look at the code…

Method #2: Table computation

```
static unsigned char table[256]; // Lookup table

void initialize_table() {
  for(int i = 0; i < 256; i++ ) {
    int cnt = 0;
    int data = i;
    while(data != 0) {
      data = data & (data-1);
      cnt++;
    }
    table[i] = cnt;
  }
}
```

Method #2: Main Method

```
int count_bits(unsigned int data) {

  const unsigned char *byte =
      (unsigned char*) &data;

  // low time-complexity!!
  return table[byte[0]] + table[byte[1]] +
         table[byte[2]] + table[byte[3]];
}
```

Bitwise Operations for Detecting Duplicates

- Inspired from the first exercise in the “Cracking the Coding Interview” book
- You have to write a function that takes an ASCII character as input, and returns true if 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 very wasteful
  - You’re wasting 7 out of 8 bits
- The book suggests using a “bit field” to save space
- Let’s do it live…
  - Hopefully we’ll encounter interesting bugs

Important Takeaways

- Bitmasks can be used for all kinds of purposes
  - Divide by powers of 2
  - Turning on, turning off, flipping specific bits
  - Swap two values
- High-level languages provide similar functionality
  - But can be less convenient than in assembly, as we saw for the “counting bit” example

Conclusion

- Let's look at some practice problems…
- Next week we'll have an in-class quiz about this module
- There is an optional homework assignment #6