Use of Bitwise Operations

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
```

Why do we care?

- Answer: Time and/or Space
- Time:
  - Some costly arithmetic operations can be done faster using bitwise operations
    - Exploiting knowledge/understanding of what it really means for a number of be written in binary
- Space:
  - Some data structures can be much more compact if one can implement them at the bit level (as opposed to the byte level)

- Let's see examples, starting with the most common time-saving "bitwise trick"…

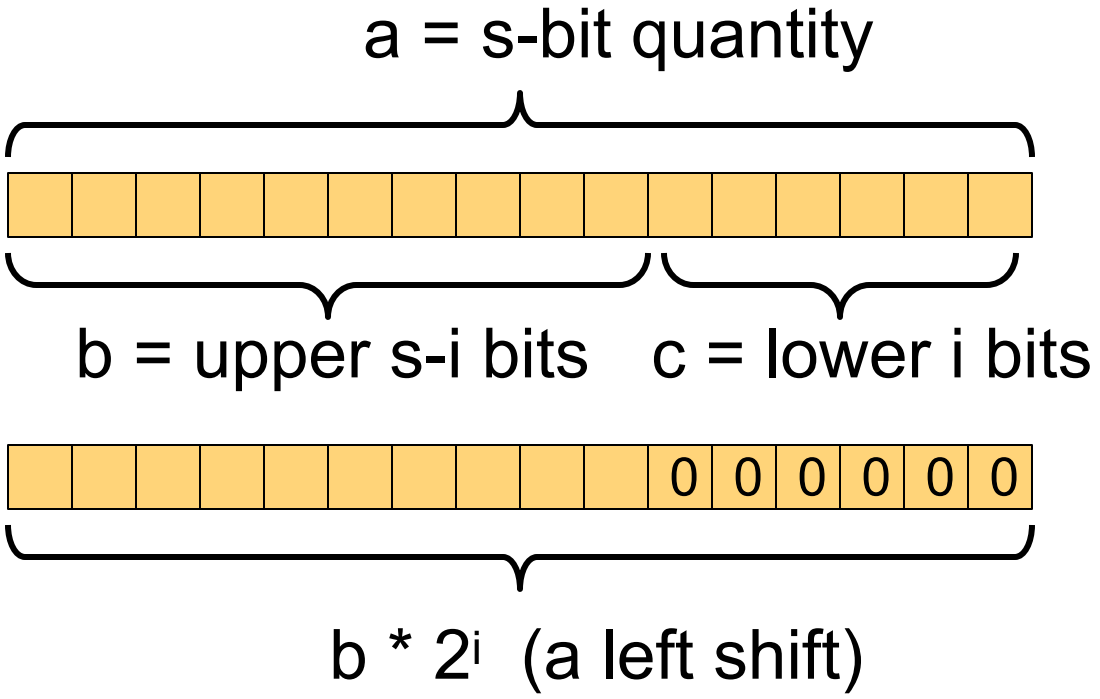
Remainder of a Division by 2^i

- Let's say we have to write a program in which we often divide integers by powers of 2 to find the remainder
- Of course, we can just compute the remainder using some mod operator ('%')
- But that mod operator performs an integer division, and division is the slowest computer arithmetic operations
- Because integers are stored in base 2, a division by a power of 2 is an "easy" case:

To find the remainder of a division of an operand by 2^i, just AND the operand by 2^i-1

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?

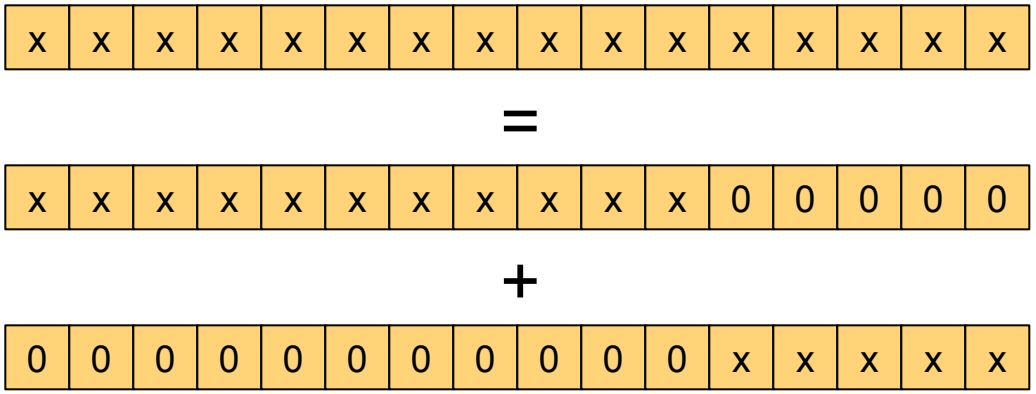


Therefore, $a = b * 2^i + c$, an c is the remainder!

The remainder is simply the lowest i bits!

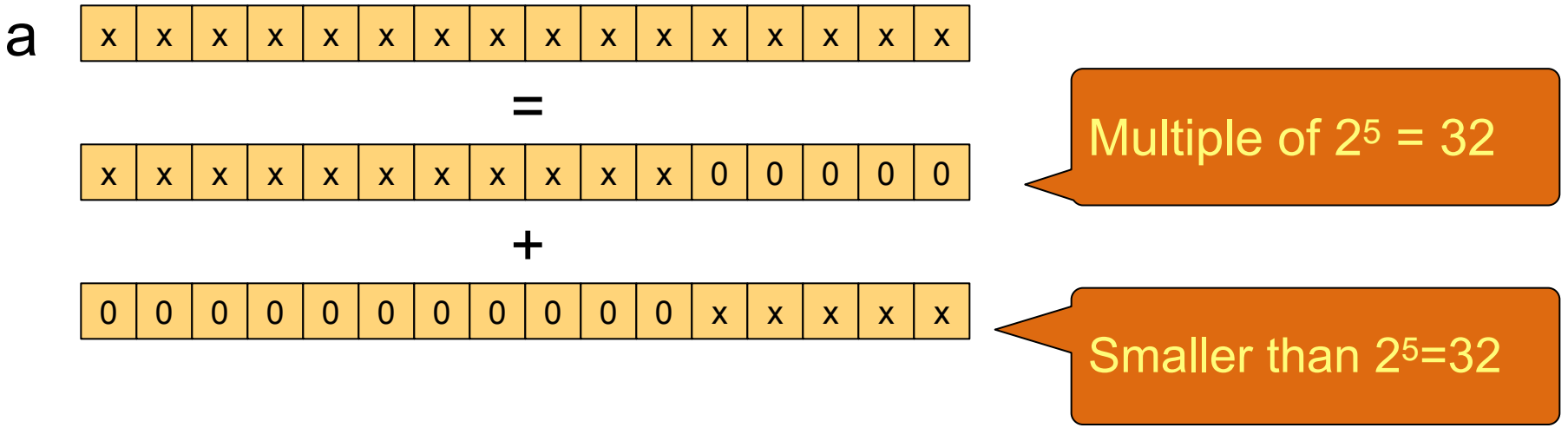
Another Way to Look at it

a



Example with i=5

Another Way to Look at it



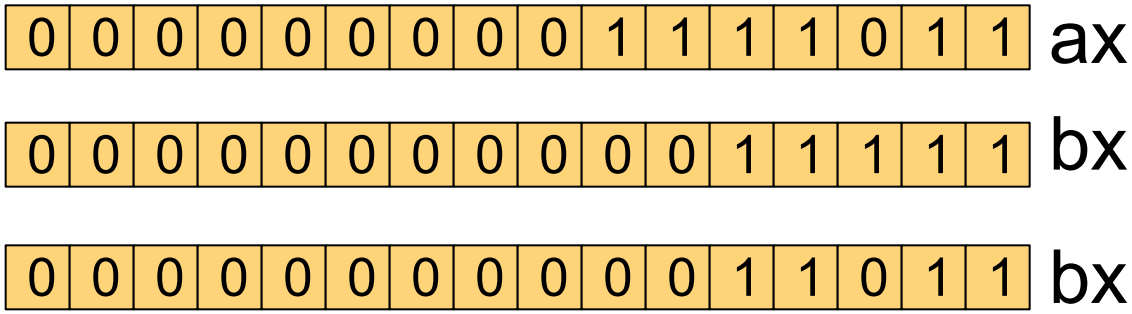
Therefore, a = <some multiple of 32> +
<some number smaller than 32>

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

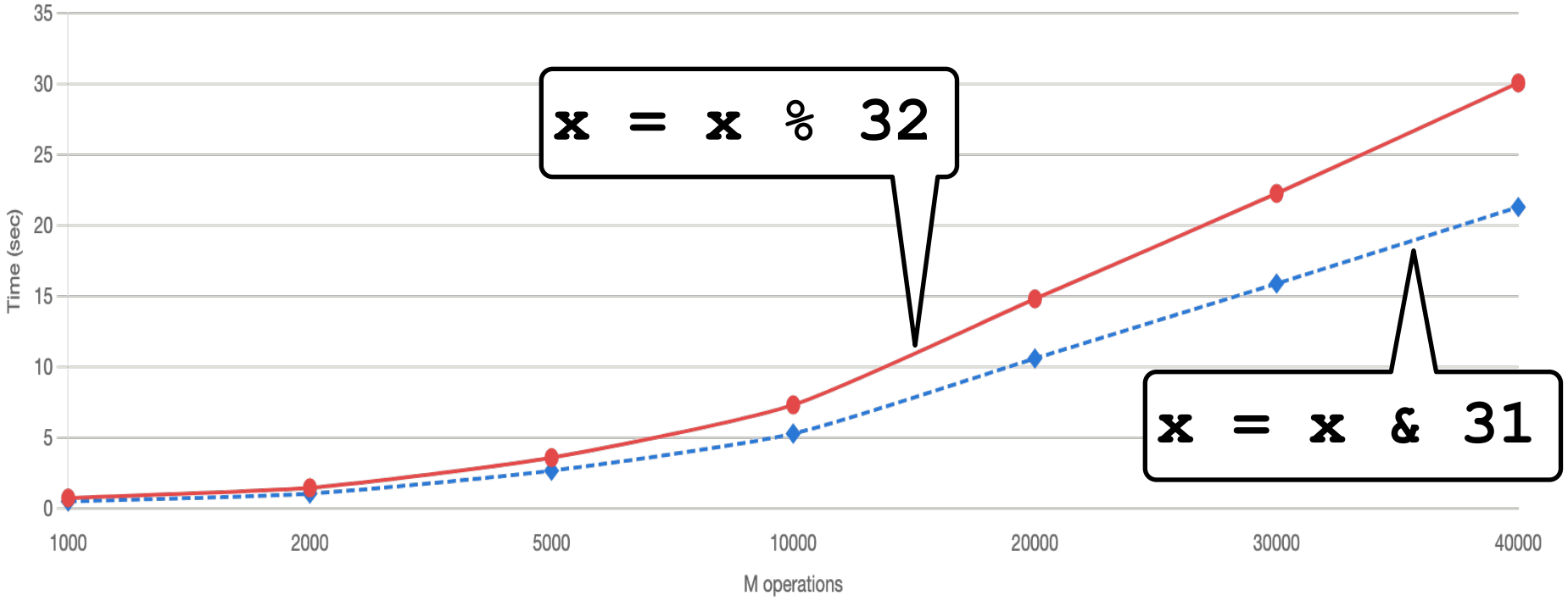
```
mov ax, 123
mov bx, 31
and bx, ax
```



- 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)

Is it Efficient?

- C++ benchmark results on my laptop



Example (live-coding)

- Using the same idea, 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

“Bitmask”

- Our example code was:

```
mov    ax, 123
mov    bx, 0001Fh ; a bitmask
and    bx, ax
```

- In the above, we do an AND with a “magical” constant: 001F
- These constants are called bitmasks
- A bitmask is a constant that we defined with bits set to 0 and 1 in the right places so that our bitwise operations modify bits of the operands in useful ways

Use of Bitmasks

- We have three operations: OR, AND, and XOR
- OR:
  - If we OR a bit with 1 we set that bit to 1 (we “turn it on”)
  - If we OR a bit with 0 we don’t change it
- AND:
  - If we AND a bit with 0 we set that bit to 0 (we “turn it off”)
  - If we AND a bit with 1 we don’t change it
- XOR:
  - If we XOR a bit with 1 we change it (we “flip it”)
  - If we XOR a bit with 0 we don’t change it

- Given the above, we can now craft bit masks and bitwise operations to modify the bits of a quantity in different ways

- The previous division by 32 example: we did an AND between our value and the bitmask 0000000000011111 to “turn off” the 11 most significant bits (AND with 0) and not change the 5 least significant bits (AND with 1)

In-Class Exercise

- Say we have some value stored in register AX
- Write assembly code that “turns on” bit 3 and “flips” bit 5 in AX (can be done in two lines of code)

In-Class Exercise

- Say we have some value stored in register AX
- Write assembly code that “turns on” bit 3 and “flips” bit 5 in AX (can be done in two lines of code)

```
or      ax, 8       ; 8d = 1000b
xor     ax, 32      ; 32d = 100000b
```

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 a bit mask "on the fly" that looks like 0…010…0
- 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
```

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!
```

Another option with rotate

- 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

Other option with a rotate shift:

```
mov  cl, bl   ; must have the bit index in cl
mov  ebx, -2   ; create a number 11111110
rol  ebx, cl ; shift it left cl times
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
```

- Its sets eax to 0, and is useful because the 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 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
  - You will see things like this all the time in disassembled code that you’re trying to reverse engineer
- Feel free to go through the example in Section 3.3, which is a good example of bit operation “craziness”

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

- Boolean Operations:
  - AND: &&
  - OR: ||
  - XOR: Not often available (Python as of 3.12 has ^)
  - 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
```

Space-Saving

- We saw examples in which time can be saved due to bitwise operations
  - Dividing by powers of 2
  - Determining the hex representation of an integer
- Let's now look at ways in which space can be saved…

Space-Saving with Bits

- Our machines are byte-addressable, so our smallest data type is one byte, yet we often need fewer than 8 bits per item in our datasets
- Example: Implementing a set of items
  - To keep track of a set of N items and we can use an array of N bool values, where the i-th value is set to True if element i is in the set
  - This should be only N bits, but because booleans are one byte, this takes up N bytes: we waste 7/8 of our memory space
- Example: Encoding items
  - Say we need to keep track of items, where each item can take only one of 4 values (e.g., “A”, “C”, “T”, “G” for a DNA sequence)
  - Since there are only 4 possible values, we need only 2 bits per item
  - But we have to use 8 bits using a byte/char data type: we waste 6/8 of our memory space
- The solution is to encode multiple items per byte!

Example: UNIX file permissions

- A very common use is for dealing with file permissions on UNIX systems, which consists of 9 booleans:
  - read / write / execute
  - owner / group / others
- Instead of using 9 bytes, file permissions are encoded on 9 bits (in a 4-byte value)
  - This avoids wasting 5 bytes for each file in the file system data structure (the i-node)
- Besides space, it's also good for time!
  - Multiple bits can be read/modified/used with a single instruction
  - As opposed to having to deal with an actual 9-byte array
- Let's see this in more detail

Using chmod from C

- In a POSIX system, there is a C library function called chmod() that modifies permissions and takes two arguments:
  - The file name
  - A 4-byte quantity: the individual permission bits
- To make life easy, the user does not have to construct the bits by hand: 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);
```

Bits to encode a Set

- Let's say we used bits to encode a set of items (True at position i means item i is present in the set)
- A common operation is: compute the size of the set
- To do this, we need to count the bits set to 1 in the representation of the set
- We have seen how to use shifts to count bits (set to one) in a value in the previous set of lecture notes in assembly
  - 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 in high-level code, let's look at the first two…

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) {

  // A cast to get to individual bytes, as we've
  // seen in a previous module!
  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…
  - Perhaps (hopefully?) we’ll encounter interesting bugs

Important Takeaways

- Bitwise operations / bitmasks can be used for all kinds of purposes
  - To save time, space, or both
- Examples we've seen:
  - Divide by powers of 2, permission bits, bits to implement sets
- All high-level languages provide bitwise operations
  - But it can be less convenient than in assembly, as we saw for the "counting bit" example
- Before we conclude…

The CLASSIC “swap two values” trick

- 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” trick

```
; 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!
```

Conclusion

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