CS 107
Lecture 4: Integer Representations and Bits / Bytes (take 3)

Wednesday, January 17, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

Reading: Reader: Bits and Bytes, Textbook: Chapter 2.2

Lecturer: Chris Gregg

Bitwise Operators

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x &amp; y</th>
<th>x</th>
<th>y</th>
<th>x ^ y</th>
<th>~x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Shifting

<table>
<thead>
<tr>
<th>x</th>
<th>x &lt;&lt; 1</th>
<th>x &lt;&lt; 2</th>
<th>x &gt;&gt; 1</th>
<th>x &gt;&gt; 2</th>
<th>x &gt;&gt; 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b10</td>
<td>0b100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0b101</td>
<td>0b1010</td>
<td>0b10100</td>
<td>0b10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>64</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
<td>60</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Today's Topics

• Logistics
  • Labs start today!
  • Assign1 released, due Wednesday Jan. 24th at 11:59pm
• Reading: Reader: Bits and Bytes, Textbook: Chapter 2.2 (very mathy…)
• Integer Representations
  • Addressing and Byte Ordering
• Boolean Algebra
• Boolean practice in gdb
We've already talked about the fact that a memory address (pointer) points to a particular byte. But, what if we want to store a data type that has more than one byte? The \texttt{int} type on our machines is 4 bytes long. So, how is a byte stored in memory?

We have choices!

First, let's talk about the ordering of the bytes in a 4-byte hex number. We can represent an \texttt{int} as 8-digit hex numbers:

\[
0x01234567
\]

We can separate out the bytes:

\[
0x \hspace{1em} 01 \hspace{1em} 23 \hspace{1em} 45 \hspace{1em} 67
\]
Addressing and Byte Ordering

<table>
<thead>
<tr>
<th>01</th>
<th>23</th>
<th>45</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0001</td>
<td>0010 0011</td>
<td>0100 0101</td>
<td>0110 0111</td>
</tr>
</tbody>
</table>

- Some machines choose to store the bytes ordered from least significant byte to most significant byte, called “little endian” (because the “little end” comes first).

- Other machines choose to store the bytes ordered from most significant byte to least significant byte, called “big endian” (because the “big end” comes first).
Addressing and Byte Ordering

• Our 0x01234567 number would look like this in memory for a little endian computer (which, by the way, is the way the myth computers store ints):

<table>
<thead>
<tr>
<th>byte:</th>
<th>67</th>
<th>45</th>
<th>23</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>address:</td>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

• A big-endian representation would look like this:

<table>
<thead>
<tr>
<th>byte:</th>
<th>01</th>
<th>23</th>
<th>45</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>address:</td>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Many times we don’t care how our integers are stored, but in cs107 we will! Let’s look at a sample program and dig under the hood to see how little-endian works.
Addressing and Byte Ordering

- Our **0x01234567** number would look like this in memory for a little endian computer (which, by the way, is the way the myth computers store ints):

  - **address:** 0x100  0x101  0x102  0x103
  - **value:** 67     45     23     01

- A big-endian representation would look like this:

  - **address:** 0x100  0x101  0x102  0x103
  - **value:** 01     23     45     67

Many times we don’t care how our integers are stored, but in cs107 we will! Let’s look at a sample program and dig under the hood to see how little-endian works.
```c
#include<stdio.h>
#include<stdlib.h>

int main() {
    // a variable
    int a = 0x01234567;

    // print the variable in big endian format
    printf("a's value: 0x%.8x\n",a);
    return 0;
}
```
$ gcc -g -O0 -std=gnu99 big_endian.c -o big_endian
$ ./big_endian
a's value: 0x01234567

$ gdb big_endian
GNU gdb (Ubuntu 7.7.1-0ubuntu5-14.04.3) 7.7.1
...
(gdb) break main
Breakpoint 1 at 0x400535: file big_endian.c, line 6.
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/107/lectures/lecture2_bits_bytes_continued/big_endian

Breakpoint 1, main () at big_endian.c:6
6    int a = 0x01234567;
(gdb) n
9    printf("a's value: 0x%08x\n",a);
(gdb) p/x a
$1 = 0x1234567
(gdb) p &a
$2 = (int *) 0x7fffffffe98c
(gdb) x/16bx &a
0x7fffffffe98c:0x67 0x45 0x23 0x01 0x00 0x00 0x00 0x00
0x7fffffffe994:0x00 0x00 0x00 0x00 0x45 0x2f 0xa3 0xf7
(gdb)

Note the ordering: 0x01234567 is stored as Little Endian!
Boolean Algebra
Because computers store values in binary, we need to learn about boolean algebra. Most of you have already studied this in some form in math classes before, but we are going to quantify it and discuss it in the context of computing and programming.

- We can define Boolean algebra over a 2-element set, 0 and 1, where 0 represents false and 1 represents true.
- The symbols are: \( \sim \) for NOT, \& for AND, \| for OR, and \( ^\wedge \) for "exclusive or," which means that if one and only one of the values is true, the expression is true.
• Be careful! There are logical analogs to some of these that you have used in C++ and other programming languages: ! (logical NOT), && (logical AND), and || (logical OR), but we are now talking about bit operations that result in 0 or 1 for each bit in a number.

• The bitwise operators use single character representations for AND and OR, not double-characters.
Boolean Algebra

- When a boolean operator is applied to two numbers (or, in the case of \(\sim\), a single number), the operator is applied to the corresponding bits in each number. For example:

\[
\begin{array}{c|c|c}
0110 & 0110 & 0110 \\
\& & \| & ^ \\
1100 & 1100 & 1100 \\
0100 & 1110 & 1010 \\
\end{array}
\]

- Full review sheet: https://107.danielr.org/review

- Bitwise review: https://107.danielr.org/problems
A common use of bit-level operations is to implement masking operations, where a mask is a bit pattern that will be used to choose a selected set of bits in a word. For example, the mask of $0xFF$ means the lowest byte in an integer. To get the low-order byte out of an integer, we simply use the bitwise AND operator with the mask:

```java
int j = 0x89ABCDEF;
int k = j & 0xFF; // k now holds the value 0xEF,
    // which is the low-order byte of j
```

A useful expression is $\sim 0$, which makes an integer with all 1s, regardless of the size of the integer.
Boolean Algebra: Bit Masking

Challenge 1: write an expression that sets the least significant byte to all ones, and all other bytes of the number (assume it is the variable \( j \)) left unchanged. E.g.

\[
0x87654321 \rightarrow 0x876543FF
\]

Possible answer: \( j \mid 0xFF \)

Challenge 2: write an expression that complements all but the least significant byte of \( j \), with the least significant byte unchanged. E.g.

\[
0x87654321 \rightarrow 0x789ABC21
\]

Possible answer: \( j \wedge \sim 0xFF \)
C provides operations to shift bit patterns to the left and to the right.

The \(<\!<\) operator moves the bits to the left, replacing the lower order bits with zeros and dropping any values that would be bigger than the type can hold:

\[ x \ll k \] will shift \( x \) to the left by \( k \) number of bits.

Examples for an 8-bit binary number:

\[ 00110111 \ll 2 \text{ returns } 11011100 \]
\[ 01100011 \ll 4 \text{ returns } 00110000 \]
\[ 10010101 \ll 4 \text{ returns } 01010000 \]
Boolean Algebra: Shift Operations

There are actually two flavors of right shift, which work differently depending on the value and type of the number you are shifting.

A *logical* right shift moves the values to the right, replacing the upper bits with 0s.

An *arithmetic* right shift moves the values to the right, replacing the upper bits with a copy of the most significant bit. This may seem weird! But, we will see why this is useful soon!

Examples for an 8-bit binary number:

**Logical right shift:**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Shift</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110111</td>
<td>&gt;&gt; 2</td>
<td>00001101</td>
</tr>
<tr>
<td>10110111</td>
<td>&gt;&gt; 2</td>
<td>00101101</td>
</tr>
<tr>
<td>01100011</td>
<td>&gt;&gt; 4</td>
<td>00000110</td>
</tr>
<tr>
<td>10010101</td>
<td>&gt;&gt; 4</td>
<td>00001001</td>
</tr>
</tbody>
</table>

Examples for an 8-bit binary number:

**Arithmetic right shift:**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Shift</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110111</td>
<td>&gt;&gt; 2</td>
<td>00001101</td>
</tr>
<tr>
<td>10110111</td>
<td>&gt;&gt; 2</td>
<td>11001101</td>
</tr>
<tr>
<td>01100011</td>
<td>&gt;&gt; 4</td>
<td>00000110</td>
</tr>
<tr>
<td>10010101</td>
<td>&gt;&gt; 4</td>
<td>11110101</td>
</tr>
</tbody>
</table>
The right-shift (>>) operator behaves differently for unsigned and signed numbers:

- **Unsigned** numbers are **logically**-right shifted (by shifting in 0s, always)

- **Signed** numbers are **arithmetically**-right shifted (by shifting in the sign bit)

```c
int main() {
    int a = 1048576;
    int a_rs8 = a >> 8;

    int b = -1048576;
    int b_rs8 = b >> 8;

    printf("a = %d:\t", a);
    show_bytes((byte_pointer) &a, sizeof(int));

    printf("a >> 8 = %d:\t", a_rs8);
    show_bytes((byte_pointer) &a_rs8, sizeof(int));

    printf("b = %d:\t", b);
    show_bytes((byte_pointer) &b, sizeof(int));

    printf("b >> 8 = %d:\t", b_rs8);
    show_bytes((byte_pointer) &b_rs8, sizeof(int));

    return 0;
}
```

$ ./right_shift
a = 1048576: 00 00 10 00
a >> 8 = 4096: 00 10 00 00
b = -1048576: 00 00 f0 ff
b >> 8 = -4096: 00 f0 ff ff

(run on a little-endian machine)
Shift Operation Pitfalls

There are two important things you need to consider when using the shift operators:

1. The C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. Almost all compilers / machines use arithmetic shifts for signed integers, and you can most likely assume this. Don't be surprised if some Internet pedant yells at you about it some day. :) All unsigned integers will always use a logical right shift (more on this later!)

2. Operator precedence can be tricky! Example:

   $1 \ll 2 + 3 \ll 4$ means this: $1 \ll (2 + 3) \ll 4$, because addition and subtraction have a higher precedence than shifts!

   Always parenthesize to be sure:
   $(1 \ll 2) + (3 \ll 4)$
Let's take a look at lots of examples:

If you want to try the examples out yourself. On myth:

```bash
$ cd CS107
$ cp -r /afs/ir/class/cs107/lecture-code/lect3 .
cd lect3
make
ls # to see the files
```

- More practice:
  - Full review sheet: https://107.danielr.org/review
  - Bitwise review: https://107.danielr.org/problems
References and Advanced Reading

• References:
  • argc and argv: http://crasseux.com/books/ctutorial/argc-and-argv.html
  • The C Language: https://en.wikipedia.org/wiki/C_(programming_language)
  • Kernighan and Ritchie (K&R) C: https://www.youtube.com/watch?v=de2Hsvxaf8M
  • C Standard Library: http://www.cplusplus.com/reference/clibrary/
  • https://en.wikipedia.org/wiki/Bitwise_operations_in_C
  • http://en.cppreference.com/w/c/language/operator_precedence

• Advanced Reading:
  • After All These Years, the World is Still Powered by C Programming
  • Is C Still Relevant in the 21st Century?
  • Why Every Programmer Should Learn C
• References:
  • Two's complement calculator: http://www.convertforfree.com/twos-complement-calculator/
  • Wikipedia on Two's complement: https://en.wikipedia.org/wiki/Two%27s_complement
  • The sizeof operator: http://www.geeksforgeeks.org/sizeof-operator-c/

• Advanced Reading:
  • Signed overflow: https://stackoverflow.com/questions/16056758/c-c-unsigned-integer-overflow
  • https://stackoverflow.com/questions/34885966/when-an-int-is-cast-to-a-short-and-truncated-how-is-the-new-value-determined
4-bit two's complement signed integer representation.
4-bit unsigned integer representation