CS 107
Lecture 3: Integer Representations and Bits / Bytes (take 2)

Friday, January 12, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

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

Lecturer: Chris Gregg
Today's Topics

- Logistics
  - Labs start next Wednesday (please put in your preferences)
  - Assign0 Due on Monday at 11:59pm
- Reading: Reader: Bits and Bytes, Textbook: Chapter 2.2 (very mathy…)
- Integer Representations
  - Signed vs Unsigned numbers
  - Casting in C
  - Signed and unsigned comparisons
  - The `sizeof` operator
  - Min and Max integer values
  - Truncating integers
  - two's complement overflow
  - More on extending the bit representation of numbers
  - Truncating numbers
- Data Sizes
- Addressing and Byte Ordering
- Boolean Algebra
Review: Practice with two's complement

Convert the following 8-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a. \(-4\) (11111100) \(\rightarrow\) 00000100
b. 27 (00011011) \(\rightarrow\) 11100101

c. \(-127\) (10000001) \(\rightarrow\) 01111111

d. 1 (00000001) \(\rightarrow\) 11111111
Casting Between Signed and Unsigned

Converting between two numbers in C can happen explicitly (using a parenthesized cast), or implicitly (without a cast):

**Explicit**
```
1 int tx, ty;
2 unsigned ux, uy;
3 ...                    \[ \text{...} \]
4 tx = (int) ux;
5 uy = (unsigned) ty;
```

**Implicit**
```
1 int tx, ty;
2 unsigned ux, uy;
3 ...                    \[ \text{...} \]
4 tx = ux;  // cast to signed
5 uy = ty;  // cast to unsigned
```

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is. You cannot convert a signed number to its unsigned counterpart using a cast!
Casting Between Signed and Unsigned

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is. You cannot convert a signed number to its unsigned counterpart using a cast!

```c
// test_cast.c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int v = -12345;
    unsigned int uv = (unsigned int) v;
    printf("v = %d, uv = %u\n", v, uv);
    return 0;
}
```

When you run the program:

```
$ ./test_cast
v = -12345, uv = 4294954951
```

Casting Between Signed and Unsigned

printf has three 32-bit integer representations:

%d : signed 32-bit int
%u : unsigned 32-bit int
%x : hex 32-bit int

As long as the value is a 32-bit type, printf will treat it according to the formatter it is applying:

```c
int x = -1;
unsigned u = 3000000000; // 3 billion

printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);
```

$ ./test_printf
x = 4294967295 = -1
u = 3000000000 = -1294967296
Signed vs Unsigned Number Wheels

4-bit
two's complement
signed integer representation

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

4-bit
unsigned integer representation

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td>Unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td>Unsigned</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: In C, 0 is false and everything else is true. When C produces a boolean value, it always chooses 1 to represent true.
Comparison between signed and unsigned integers

Let's try some more...a bit more abstractly.

```c
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

Which many of the following statements are true? (assume that variables are set to values that place them in the spots shown)

- `s3 > u3`
- `u2 > u4`
- `s2 > s4`
- `s1 > s2`
- `u1 > u2`
- `s1 > u3`
Comparison between signed and unsigned integers

Let's try some more...a bit more abstractly.

```c
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

Which many of the following statements are true? (assume that variables are set to values that place them in the spots shown)

- `s3 > u3` : true
- `u2 > u4` : true
- `s2 > s4` : false
- `s1 > s2` : true
- `u1 > u2` : true
- `s1 > u3` : true
The sizeof Operator

As we have seen, integer types are limited by the number of bits they hold. On the 64-bit myth machines, we can use the `sizeof` operator to find how many bytes each type uses:

```c
int main() {
    printf("sizeof(char): %d\n", (int) sizeof(char));
    printf("sizeof(short): %d\n", (int) sizeof(short));
    printf("sizeof(int): %d\n", (int) sizeof(int));
    printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
    printf("sizeof(long): %d\n", (int) sizeof(long));
    printf("sizeof(long long): %d\n", (int) sizeof(long long));
    printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
    printf("sizeof(void *): %d\n", (int) sizeof(void *));
    return 0;
}
```

```
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Width in bytes</th>
<th>Width in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>size_t</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>void *</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>
MIN and MAX values for integers

Because we now know how bit patterns for integers works, we can figure out the maximum and minimum values, designated by `INT_MAX`, `UINT_MAX`, `INT_MIN`, (etc.), which are defined in `limits.h`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Width (bytes)</th>
<th>Width (bits)</th>
<th>Min in hex (name)</th>
<th>Max in hex (name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
<td>80 (CHAR_MIN)</td>
<td>7F (CHAR_MAX)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>FF (UCHAR_MAX)</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
<td>8000 (SHRT_MIN)</td>
<td>7FFF (SHRT_MAX)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>FFFF (USHRT_MAX)</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
<td>80000000 (INT_MIN)</td>
<td>7FFFFFFFFF (INT_MAX)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>FFFFFFFFFF (UINT_MAX)</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
<td>8000000000000000 (LONG_MIN)</td>
<td>7FFFFFFFFFFFFFFFFF (LONG_MAX)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td>64</td>
<td>0</td>
<td>FFFFFFFFFFFFFFFFFF (ULONG_MAX)</td>
</tr>
</tbody>
</table>
Sometimes we want to convert between two integers having different sizes. E.g., a **short** to an **int**, or an **int** to a **long**.

We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a smaller data type to a bigger data type.

This is easy for unsigned values: simply add leading zeros to the representation (called "zero extension").

```c
unsigned short s = 4;
// short is a 16-bit format, so                     s = 0000 0000 0000 0100b
unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```
Expanding the bit representation of a number

For signed values, we want the number to remain the same, just with more bits. In this case, we perform a "sign extension" by repeating the sign of the value for the new digits. E.g.,

```java
short s = 4;
// short is a 16-bit format, so s = 0000 0000 0000 0100b
int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b

— or —

short s = -4;
// short is a 16-bit format, so s = 1111 1111 1111 1100b
int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```
// sign-extension example

int main() {
    short sx = -12345;       // -12345
    unsigned short usx = sx; // 53191
    int x = sx;              // -12345
    unsigned ux = usx;       // 53191
    printf("sx = %d:\t", sx);
    printf("usx = %u:\t", usx);
    printf("x  = %d:\t", x);
    printf("ux = %u:\t", ux);
    return 0;
}

$ ./sign_extension
sx = -12345: c7 cf
usx = 53191: c7 cf
x  = -12345: c7 cf ff ff
ux = 53191: c7 cf 00 00

careful: this was printed on the little-endian myth machines!
What if we want to reduce the number of bits that a number holds? E.g.

```java
int x = 53191;
short sx = (short) x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit `int`), 53191:

```
0000 0000 0000 0000 1100 1111 1100 0111
```

When we cast `x` to a short, it only has 16-bits, and C truncates the number:

```
1100 1111 1100 0111
```

What is this number in decimal? Well, it must be negative (b/c of the initial 1), and it is `-12345`. 
What if we want to reduce the number of bits that a number holds? E.g.

```java
int x = 53191;        // 53191
short sx = (short) x; // -12345
int y = sx;
```

This is a form of overflow! We have altered the value of the number. Be careful!

We don't have enough bits to store the int in the short for the value we have in the int, so the strange values occur.

What is y above? We are converting a short to an int, so we sign-extend, and we get -12345!

```
1100 1111 1100 0111 becomes
1111 1111 1111 1111 1100 1111 1100 0111
```

Truncating Numbers: Signed

If the number does fit into the smaller representation in the current form, it will convert just fine.

```c
int x = -3;        // -3
short sx = (short) -3; // -3
int y = sx;       // -3
```

x:      1111 1111 1111 1111 1111 1111 1111 1101 becomes
sx:     1111 1111 1111 1111 1101

Play around here: http://www.convertforfree.com/twos-complement-calculator/
We can also lose information with unsigned numbers:

```c
unsigned int x = 128000;
unsigned short sx = (short) x;
unsigned int y = sx;
```

Bit representation for \( x = 128000 \) (32-bit unsigned int):

```
0000 0000 0000 0001 1111 0100 0000 0000
```

Truncated unsigned short \( sx \):

```
1111 0100 0000 0000
```

which equals 62464 decimal.

Converting back to an unsigned int, \( y = 62464 \)
# include<stdio.h>
#include<stdlib.h>
#include<limits.h>  // for UINT_MAX

int main() {
    unsigned int a = UINT_MAX;
    unsigned int b = 1;
    unsigned int c = a + b;

    printf("a = %u\n",a);
    printf("b = %u\n",b);
    printf("a + b = %u\n",c);

    return 0;
}

$ ./unsigned_overflow
a = 4294967295
b = 1
a + b = 0

Technically, unsigned integers in C don't overflow, they just wrap. You need to be aware of the size of your numbers. Here is one way to test if an addition will fail:

// for addition
#include <limits.h>
unsigned int a = <something>;
unsigned int x = <something>;
if (a > UINT_MAX - x) /* `a + x` would overflow */;
Signed overflow wraps around to the negative numbers:

YouTube fell into this trap — their view counter was a signed, 32-bit int. They fixed it after it was noticed, but for a while, the view count for Gangnam Style (the first video with over \texttt{INT\_MAX} number of views) was negative.
Overflow in Signed Addition

In the news on January 5, 2022 (!):

https://arstechnica.com/gadgets/2022/01/google-fixes-nightmare-android-bug-that-stopped-user-from-calling-911/
Signed overflow wraps around to the negative numbers.

Signed over
flow wraps around to the negative numbers.

Technically, signed integers in C produce undefined behavior when they overflow. On two's complement machines (virtually all machines these days), it does overflow predictably. You can test to see if your addition will be correct:

```c
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for INT_MAX

int main() {
    int a = INT_MAX;
    int b = 1;
    int c = a + b;

    printf("a = %d\n",a);
    printf("b = %d\n",b);
    printf("a + b = %d\n",c);
    return 0;
}
```

```bash
$ ./signed_overflow
a = 2147483647
b = 1
a + b = -2147483648
```

Technically, signed integers in C produce undefined behavior when they overflow. On two's complement machines (virtually all machines these days), it does overflow predictably. You can test to see if your addition will be correct:

```c
// for addition
#include <limits.h>
int a = <something>;
int x = <something>;
if ((x > 0) && (a > INT_MAX - x)) /* `a + x` would overflow */;
if ((x < 0) && (a < INT_MIN - x)) /* `a + x` would underflow */;
```
Data Sizes
We found out above that on the myth computers, the `int` representation is comprised of 32-bits, or four 8-bit bytes. But the C language does not mandate this. To the right is Figure 2.3 from your textbook:

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Signed</th>
<th>Unsigned</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>[signed] char</td>
<td>char</td>
<td>unsigned char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>unsigned</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>unsigned long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>int32_t</td>
<td>int32_t</td>
<td>uint32_t</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int64_t</td>
<td>int64_t</td>
<td>uint64_t</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>char</td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
There are guarantees on the lower-bounds for type sizes, but you should expect that the myth machines will have the numbers in the 64-bit column.

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32-bit</td>
</tr>
<tr>
<td>[signed] char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
</tr>
<tr>
<td>int32_t</td>
<td>4</td>
</tr>
<tr>
<td>int64_t</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>
You can be guaranteed the sizes for `int32_t` (4 bytes) and `int64_t` (8 bytes)

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Signed</th>
<th>Unsigned</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>[signed] char</td>
<td>signed char</td>
<td>unsigned char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>signed short</td>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>signed</td>
<td>unsigned</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>signed long</td>
<td>unsigned long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
We briefly mentioned *unsigned* types on the first day of class. These are integer types that are strictly positive.

By default, integer types are signed.
C allows a variety of ways to order keywords to define a type. The following all have the same meaning:

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signed</strong></td>
<td>Unsigned</td>
</tr>
<tr>
<td>[signed] char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>int</td>
<td>unsigned</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
</tr>
<tr>
<td>char *</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
On the myth machines, pointers are 64-bits long, meaning that a program can "address" up to $2^{64}$ bytes of memory, because each byte is individually addressable.

This is a lot of memory! It is 16 exabytes, or $1.84 \times 10^{19}$ bytes. Older, 32-bit machines could only address $2^{32}$ bytes, or 4 Gigabytes.

64-bit machines can address 4 billion times more memory than 32-bit machines...

Machines will not need to address more than $2^{64}$ bytes of memory for a long, long time.
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:

\[
0x \quad 01 \quad 23 \quad 45 \quad 67
\]
Addressing and Byte Ordering

- 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>Address</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>67</td>
</tr>
<tr>
<td>0x101</td>
<td>45</td>
</tr>
<tr>
<td>0x102</td>
<td>23</td>
</tr>
<tr>
<td>0x103</td>
<td>01</td>
</tr>
</tbody>
</table>

• A big-endian representation would look like this:

<table>
<thead>
<tr>
<th>Address</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>01</td>
</tr>
<tr>
<td>0x101</td>
<td>23</td>
</tr>
<tr>
<td>0x102</td>
<td>45</td>
</tr>
<tr>
<td>0x103</td>
<td>67</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):

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

• A big-endian representation would look like this:

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

```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;
}
```
Addressing and Byte Ordering

$ 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: `~` for NOT, `&` for AND, `|` for OR, and `^` 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 ~, a single number), the operator is applied to the corresponding bits in each number. For example:

```
  0110 & 1100 ---- 0100
  0110 | 1100 ---- 1110
  0110 ^ 1100 ---- 1010
  ~ 1100 ---- 0011
```
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:

```c
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 `~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: \( j \uparrow \lnot 0xFF \)
Boolean Algebra: Shift Operations

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:

\[
\begin{align*}
00110111 \ll 2 & \text{ returns } 11011100 \\
01100011 \ll 4 & \text{ returns } 00110000 \\
10010101 \ll 4 & \text{ returns } 01010000
\end{align*}
\]
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:**

- \(00110111 >> 2\) returns \(00001101\)
- \(10110111 >> 2\) returns \(00101101\)
- \(01100011 >> 4\) returns \(00000110\)
- \(10010101 >> 4\) returns \(00001001\)

Examples for an 8-bit binary number:

**Arithmetic right shift:**

- \(00110111 >> 2\) returns \(00001101\)
- \(10110111 >> 2\) returns \(11101101\)
- \(01100011 >> 4\) returns \(00000110\)
- \(10010101 >> 4\) returns \(11111001\)
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:

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

   Always parenthesize to be sure:
   \[ (1 \ll 2) + (3 \ll 4) \]
Practice!

Let's take a look at lots of examples:

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

```
$ 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:
  • 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 and Advanced Reading

• 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