CS 107
Lecture 2: Bits and Bytes (continued)

Friday, September 29, 2017

Computer Systems
Fall 2017
Stanford University
Computer Science Department

Reading: Chapter 2.1

Lecturers: Julie Zelenski and Chris Gregg
Today's Topics

• Assign0 -- Piazza
• Recap: Binary, Hexadecimal, Conversions
• \texttt{argc} and \texttt{argv}
• Data Sizes
• Addressing and Byte Ordering
• Representing Strings
• Boolean Algebra
In C, we write a hexadecimal with a starting `0x`. So, you will see numbers such as `0xf1d37b`, which means that it is a hex number.

You should memorize the binary representations for each hex digit. One trick is to memorize A (1010), C (1100), and F (1111), and the others are easy to figure out.

Let's practice some hex to binary and binary to hex conversions:

### Convert: 0x28AF94 to binary.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>2</th>
<th>8</th>
<th>A</th>
<th>F</th>
<th>9</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0010</td>
<td>1000</td>
<td>1010</td>
<td>1111</td>
<td>1001</td>
<td>0100</td>
</tr>
</tbody>
</table>

0x28AF94 is binary 0b1010001010111110010100
Convert: $0b1011111111101010001111$ to hexadecimal.

<table>
<thead>
<tr>
<th>Binary</th>
<th>10</th>
<th>1111</th>
<th>1111</th>
<th>1010</th>
<th>1000</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>2</td>
<td>F</td>
<td>F</td>
<td>A</td>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

$0b1011111111101010001111$ is hexadecimal $2FFA8F$
Decimal to Hexadecimal

To convert from decimal to hexadecimal, you need to repeatedly divide the number in question by 16, and the remainders make up the digits of the hex number:

314156 decimal:

314,156 / 16 = 19,634 with 12 remainder: C
19,634 / 16 = 1,227 with 2 remainder: 2
1,227 / 16 = 76 with 11 remainder: B
76 / 16 = 4 with 12 remainder: C
4 / 16 = 0 with 4 remainder: 4

Reading from bottom up: 0x4CB2C
Hexidecimal to Decimal

To convert from hexadecimal to decimal, multiply each of the hexadecimal digits by the appropriate power of 16:

0x7AF:

\[
7 \times 16^2 + 10 \times 16 + 15
= 7 \times 256 + 160 + 15
= 1792 + 160 + 15 = 1967
\]
Let the computer do it!

Honestly, hex to decimal and vice versa are easy to let the computer handle. You can either use a search engine (Google does this automatically), or you can use a python one-liner:

```
cgregg@myth10:~$ python -c "print(hex(314156))"
0x4cb2c

cgregg@myth10:~$ python -c "print(0x7af)"
1967

cgregg@myth10:~$
```
Let the computer do it!

You can also use Python to convert to and from binary:

(cgregg@myth10:~)$ python -c "print(bin(0x173A4C))"
0b1011110011010010011000
(cgregg@myth10:~)$ python -c "print(hex(0b11110101011011011011))"
0x3cadb3
(cgregg@myth10:~)$

(but you should memorize this as it is easy and you will use it frequently)
In assignment 0, you modify the triangle.c program, which has the following main() function:

```c
int main(int argc, char *argv[]) {
    int nlevels = 3;
    print_triangle(nlevels);
    return 0;
}
```
In assignment 0, you modify the triangle.c program, which has the following main() function:

```c
int main(int argc, char *argv[])
{
    int nlevels = 3;
    print_triangle(nlevels);
    return 0;
}
```
In assignment 0, you modify the triangle.c program, which has the following main() function:

```c
int main(int argc, char *argv[]) {
    int nlevels = 3;
    print_triangle(nlevels);
    return 0;
}
```

`argc` and `argv` are the "command line arguments":

- `int argc` is the number of space-separated arguments, including the program name.
- `char *argv[]` is a pointer to an array of character strings that contain the arguments.
**argc and argv**

`argc` and `argv` are the "command line arguments":

- `int argc` is the number of space-separated arguments, *including the program name*.
- `char *argv[]` is a pointer to an array of character strings that contain the arguments.

Let's look at the following program:

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

int main(int argc, char *argv[]) {
    for (int i=0; i < argc; i++) {
        printf("Argument %d: %s\n",i,argv[i]);
    }
    return 0;
}
```
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char *argv[]) {
   for (int i=0; i < argc; i++) {
      printf("Argument %d: %s\n",i,argv[i]);
   }
   return 0;
}

$ ./argc_argv this is a sentence
Argument 0: ./argc_argv
Argument 1: this
Argument 2: is
Argument 3: a
Argument 4: sentence
#(argc and argv)

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

int main(int argc, char *argv[]) {
    for (int i=0; i < argc; i++) {
        printf("Argument %d: %s\n",i,argv[i]);
    }
    return 0;
}
```

$ ./argc_argv this is a sentence
Argument 0: ./argc_argv
Argument 1: this
Argument 2: is
Argument 3: a
Argument 4: sentence

argv[0] is the program name
argv[1] is the first real argument
argv[2] is the second real argument
etc.
argv is a pointer to an array of character strings, which are each pointers to null-terminated character arrays. argv[i] refers to the i^{th} array.

In memory:
argv and argv

You may also see it this way, which is exactly the same:

```
char **argv
```

argv is a pointer to an array of character strings, which are each pointers to null-terminated character arrays. argv[i] refers to the ith array.

In memory:
Data Sizes
Data Sizes

We found out above that on the myth computers, the \texttt{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>Unsigned</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed</td>
<td>Signed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[signed] char</td>
<td>unsigned char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>unsigned</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
### Data Sizes

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>Unsigned</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>[signed] char</td>
<td>unsigned char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>unsigned</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
You can be guaranteed the sizes for `int32_t` (4 bytes) and `int64_t` (8 bytes)
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.

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed</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>
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>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>[signed] char</td>
<td>short</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>long unsigned</td>
<td></td>
<td>int32_t</td>
</tr>
<tr>
<td>long unsigned int</td>
<td>int64_t</td>
<td>uint32_t</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>int32_t</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int64_t</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Addressing and Byte Ordering

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.
Addressing and Byte Ordering

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} with as 2-digit hex numbers:

\begin{align*}
0x01234567
\end{align*}

We can separate out the bytes:

\begin{align*}
0x & 01 \quad 23 \quad 45 \quad 67
\end{align*}
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):

  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.
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.
Addressing and Byte Ordering

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

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

    // print the variable
    printf("array[0]'s value: 0x%08x\n",array[0]);
    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 0xa3 0xf7
(gdb)

Note the ordering: 0x01234567 is stored as Little Endian!
Addressing and Byte Ordering

$ gcc -g -Og -std=gnu99 intRepresentation.c -o intRepresentation
$ ./intRepresentation
array[0]'s value: 0x01234567
$ gdb intRepresentation
...
(gdb) break main
Breakpoint 1 at 0x40059d: file intRepresentation.c, line 4.
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/107/lecture1_bits_bytes/intRepresentation

Breakpoint 1, main () at intRepresentation.c:4
4   int main() {
(gdb) n
6       int *array = malloc(sizeof(int));
(gdb) n
8       array[0] = a;
(gdb) n
10      printf("array[0]'s value: 0x%08x\n",array[0]);
(gdb) p/x array[0]
$2 = 0x1234567
(gdb) x/16bx array
0x602010:   0x67    0x45    0x23    0x01    0x00    0x00    0x00    0x00
0x602018:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
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 \textit{false} and 1 represents \textit{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.
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:

\[
\begin{array}{c}
\text{\&} \\
0110 \\
1100 \\
\hline
0110 \\
\end{array} \quad \begin{array}{c}
\mid \\
1100 \\
\hline
1010 \\
\end{array} \quad \begin{array}{c}
\wedge \\
0110 \\
1100 \\
\hline
1010 \\
\end{array} \quad \begin{array}{c}
\sim \\
1100 \\
\hline
0011 \\
\end{array}
\]
Boolean Algebra: Mystery Function

• Let's look at a mystery function!

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

void mystery(int *x, int *y) {
    if (x != y) {
        *y = *x ^ *y;
        *x = *x ^ *y;
        *y = *x ^ *y;
    }
}

int main(int argc, char *argv[]) {
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    printf("x:%d, y:%d\n",x,y);
    mystery(&x,&y);
    printf("x:%d, y:%d\n",x,y);
    return 0;
}
```

$ ./mystery 4 5
Boolean Algebra: Mystery Function

• Let's look at a mystery function!

```
#include<stdlib.h>
#include<stdio.h>

void mystery(int *x, int *y) {
    if (x != y) {
        *y = *x ^ *y;
        *x = *x ^ *y;
    }
}

int main(int argc, char *argv[]) {
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    printf("x:%d, y:%d\n",x,y);
    mystery(&x,&y);
    printf("x:%d, y:%d\n",x,y);
    return 0;
}
```

$ ./mystery 4 5
x:4, y:5
x:5, y:4


This relies on the fact that $x^x == 0$, and the associativity and commutativity of the exclusive or function.

Incidentally, if you XOR a number with all 1s, you get the complement!
We can represent finite sets with bit vectors, where we can perform set functions such as union, intersection, and complement. For example:

bit vector $a = [01101001]$ encodes the set $A = \{0,3,5,6\}$ (reading the 1 positions from right to left, with #0 being the right-most, #7 being the left-most)

bit vector $b = [01010101]$ encodes the set $B = \{0,2,4,6\}$

The $|$ operator produces a set union:

$a \mid b \rightarrow [01111101]$, or $A \cup B = \{0,2,3,4,5,6\}$

The $\&$ operator produces a set intersection:

$a \& b \rightarrow [01000001]$, or $A \cap B = \{0,6\}$
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 x = 0x89ABCDEF;
int y = x & 0xFF; // y now holds the value 0xEF, which is the low-order byte of x
```

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 x left unchanged. E.g.

\[ 0x87654321 \rightarrow 0x876543FF \]

Possible answer: \[ x \mid 0xFF \]

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

\[ 0x87654321 \rightarrow 0x789ABC21 \]

Possible answer: \[ x \uparrow \sim 0xFF \]
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:

\[ x \ll k \text{ will shift } x \text{ to the left by } k \text{ number of bits.} \]

Examples for an 8-bit binary number:

\[
\begin{align*}
00110111 \ll 2 & \quad \text{returns} \quad 11011100 \\
01100011 \ll 4 & \quad \text{returns} \quad 00110000 \\
10010101 \ll 4 & \quad \text{returns} \quad 01010000
\end{align*}
\]
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:
- $00110111 \gg 2$ returns $00001101$
- $10110111 \gg 2$ returns $00101101$
- $01100011 \gg 4$ returns $00000110$
- $10010101 \gg 4$ returns $00001001$

Arithmetic right shift:
- $00110111 \gg 2$ returns $00001101$
- $10110111 \gg 2$ returns $11101101$
- $01100011 \gg 4$ returns $00000110$
- $10010101 \gg 4$ returns $11111001$
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:

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

Always parenthesize to be sure:

$(1 << 2) + (3 << 4)$
Integer Representations
The C language has two different ways to represent numbers, unsigned and signed:

**unsigned**: can only represent non-negative numbers

**signed**: can represent negative, zero, and positive numbers

We are going to talk about these representations, and also about what happens when we expand or shrink an encoded integer to fit into a different type (e.g., `int` to `long`)
# Integral Data Types

What you can expect on the myth machines

<table>
<thead>
<tr>
<th>C data type</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>[signed] char</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>unsigned long</td>
<td>0</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>int32_t</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>uint32_t</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>int64_t</td>
<td>-9,223,372,036,854,775,808</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>uint64_t</td>
<td>0</td>
<td>18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

*Figure 2.10* Typical ranges for C integral data types for 64-bit programs.
So far, we have talked about converting from decimal to binary and vice-versa, which is a nice one-to-one relationship between the decimal number and its binary representation. Examples:

- \texttt{0b0001} = 1
- \texttt{0b0101} = 5
- \texttt{0b1011} = 11
- \texttt{0b1111} = 15

The range of an unsigned number is $0 \rightarrow 2^w - 1$, where $w$ is the number of bits in our integer. For example, a 32-bit int can represent numbers from $0$ to $2^{32} - 1$, or $0$ to $4,294,967,295$ (as seen in the figure on the previous slide).
Signed Integers: How do we represent them?

What if we want to represent negative numbers? We have choices!

One way we could encode a negative number is simply to designate some bit as a "sign" bit, and then interpret the rest of the number as a regular binary number and then apply the sign. For instance, for a four-bit number:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1</td>
<td>1001</td>
<td>-1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>1010</td>
<td>-2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>1011</td>
<td>-3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1101</td>
<td>-5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1110</td>
<td>-6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1111</td>
<td>-7</td>
</tr>
</tbody>
</table>

This might be okay...but we've only represented 14 of our 16 available numbers...
Signed Integers: How do we represent them?

0 001 = 1  1 001 = -1  This might be okay...but we've only
0 010 = 2  1 010 = -2  represented 14 of our 16 available numbers.
0 011 = 3  1 011 = -3
0 100 = 4  1 100 = -4  What about 0 000 and 1 000? What should
0 101 = 5  1 101 = -5  they represent?
0 110 = 6  1 110 = -6
0 111 = 7  1 111 = -7  Well...this is a bit tricky...we will get to this
                                  next time!
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