

---

# Data Representation I

Bits and Bytes

Connor Baker



Department of  
**Computer Science**

Compiled on September 15, 2019 at 10:01am

## Reading Track

This Lecture covers Chapter 2.1 (Bits and Integers)

Reading Homework:

- Read through Chapter 2.2-2.3 (Integer Representations)
  - The next lecture covers Integer Representations and Arithmetic
- Continue to Review C (This Friday's Recitation is on C). This semester, you'll be doing the following operations in Assembly Language. If you know these well in C, it will be a lot easier over the next two months:
  - Passing by reference and working with pointers in functions
  - Dereferencing values: `*ptr = 42;`
  - Pointer arithmetic with arrays: `*(ary + 3) = 42;`
  - Working with `structs` and dynamic memory
  - Working with linked lists using dynamically allocated `structs`

## Lecture Overview from CS262

Bits and Ints	Convert Between Hex, Decimal, and Binary	Convert Dec->Hex, Dec->Binary, Hex->Binary, etc.	2.1.1
	Interpret Data given the Endianness of a System	Describe Big and Little Endian	2.1.3
		Convert Big Endian Ordering to a Value	
		Convert Little Endian Ordering to a Value	
		Identify Network Byte Order Endianness	
		Identify Intel Byte Order Endianness	
	Describe the Mapping between C Code and Machine Code	Describe Machine Code	2.1.5 (+ 3.1)
		Describe how a Machine sees a Program	
		Describe why Compiled Code is not Portable	
	Describe the Four Key Boolean Algebra Operations	Describe and Draw a Truth Table for Boolean AND	2.1.6
		Describe and Draw a Truth Table for Boolean OR	
		Describe and Draw a Truth Table for Boolean XOR	
		Describe and Draw a Truth Table for Boolean Negation	
	Perform Bitwise and Logical Operations in C	Perform a Bitwise AND on two Integers	2.1.7, 2.1.8
		Perform a Bitwise OR on two Integers	
		Perform a Bitwise XOR on two Integers	
		Perform a Bitwise Negation on two Integers	
		Apply a Bit-Mask on to an Integer	
		Use a Bit-Mask to Extract Data from an Integer	
		Perform a Logical AND on two Statements	
		Perform a Logical OR on two Statements	
		Perform a Logical NOT on two Statements	
		Normalize an Integer using Logical NOT	
	Perform Arithmetic Operations on Integers by Constants	Perform Left and Right Shifts on an Integer	2.1.9

**Figure 1:** Lecture overview.

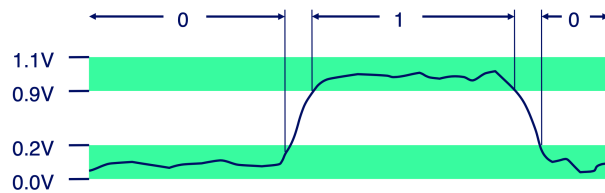
## Everything in a system is just bits

Each bit is a zero or a one. By encoding and interpreting sets of bits in various ways:

- Computers determine what to do (operations and instructions)

- Computers represent and manipulate data

Why do we use bits? Well, because they're easy to store in bistable elements (like electronics).



**Figure 2:** A high voltage represents a one, while a low voltage represents a zero. It gets fuzzy in between, but that's a problem for the Computer Engineers.

## Number Systems

A *base* (or *radix*) is a mathematical building block used to describe a number system whose digits are used to represent numbers. It (as far as whole-number bases go) is the number of numerals allowed in a base, and it includes zero.

The *maximum numeral* in a base is one less than the base.

The *radix point* is a point used to separate the integer part of a number from the fractional part.

A  $\beta$ -Expansion is a means of rewriting a number as the digits multiplied by the base which is raised to the power of position of the digit. Some examples include:

- $125_{10} = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$
- $A5.E_{16} = 10 \times 16^1 + 5 \times 16^0 + 14 \times 16^{-1}$
- $10_2 = 1 \times 2^1 + 0 \times 2^0$

The general formula is

$$\sum_{i=0}^{\omega-1} x_i \times \beta^i$$

where  $x_i$  is the  $i$ th-digit and  $\beta$  is the base.

## Converting to a base

The simplest way to compute a number in a different base is to repeatedly divide the number you want to convert by the base you want to convert, making a list of the remainders until you hit a quotient of zero. At that point, read the list in reverse and you've got your number!

Quotient	Remainder	List
123 / 2 = 61	1	1
61 / 2 = 30	1	11
30 / 2 = 15	0	110
15 / 2 = 7	1	1101
7 / 2 = 3	1	11011
3 / 2 = 1	1	110111
1 / 2 = 0	1	1101111

Reversing the list yields  $1111011_2$ , which is correct! We can verify this with a  $\beta$ -Expansion:

$$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 123.$$

### Converting between powers of bases

When converting between powers of bases, we can group digits, convert them individually, and concatenate the result. As an example, let's convert  $1101011101_2$  to hexadecimal. Since  $16 = 2^4$ , we group the binary digits into buckets of size four. This grouping process starts at the right and continues left, padding with zeros on the left if the last bucket does not contain four digits.

0011    0101    1101

Converting each individually yields

3    5    D

Concatenating the result yields  $35D_{16}$  which is correct.

This process also works in reverse. Using the same number, let's convert back to binary. Since  $16 = 2^4$ , each hexadecimal digit will produce four binary digits.

3    5    D

0011    0101    1101

Concatenating the result yields  $1101011101_2$ , which is correct.

### Prefixes for bases in C

To write a number in *decimal* in C is simple: write the number.

- `int c = 255; //255 as a decimal number`

To write a number in *binary* in C, we can take advantage of an extension GCC provides and use the `0b` prefix..

- `int c = 0b11111111; //255 as a binary number`

To write a number in *octal* in C, we can take advantage of a prefix provided by the language (`0`).

- `int c = 0377; //255 as an octal number`

To write a number in *hexadecimal* in C, we can take advantage of a prefix provided by the language (`0x`).

- `int c = 0xFF; //255 as a hexadecimal number`

### Arithmetic in different bases

As long as the base is a whole number base, arithmetic is performed exactly the same way that it is with decimal numbers.

### Byte-Oriented Memory Organization

- Conceptually, memory is just a very, very large array of bytes
  - In reality, it's become *way* more complicated but we won't touch on that until much later this semester
- An address is an index into that array of bytes
- A pointer is a variable whose value is a memory address

Note: Operating Systems give each process a Private Address Space

- A process is just a program which is being executed
- This separation protects processes from malicious actors

### Word Size Data Types

C Data Type	Typical 32-bit Byte Count	Typical 64-bit Byte Count	Zeus
<code>char</code>	1	1	1
<code>short</code>	2	2	2
<code>int</code>	4	4	4
<code>long</code>	4	8	8
<code>float</code>	4	4	4
<code>double</code>	8	8	8
<code>long double</code>	N/A	10/16	16
Pointers	4	8	8

We can guarantee that the integer type sizes are consistent across architectures by using `stdint.h`.

### `stdint.h`

The header file `stdint.h` defines the following integer types with the associated sizes:

Signed?	Standard Data Type	32-bit Byte Count	64-bit Byte Count
Yes	<code>int8_t</code>	1	1
No	<code>uint8_t</code>	1	1
Yes	<code>int16_t</code>	2	2
No	<code>uint16_t</code>	2	2
Yes	<code>int32_t</code>	4	4
No	<code>uint32_t</code>	4	4
Yes	<code>int64_t</code>	8	8
No	<code>uint64_t</code>	8	8

There are other fun things in that header, but these are all that we'll cover.

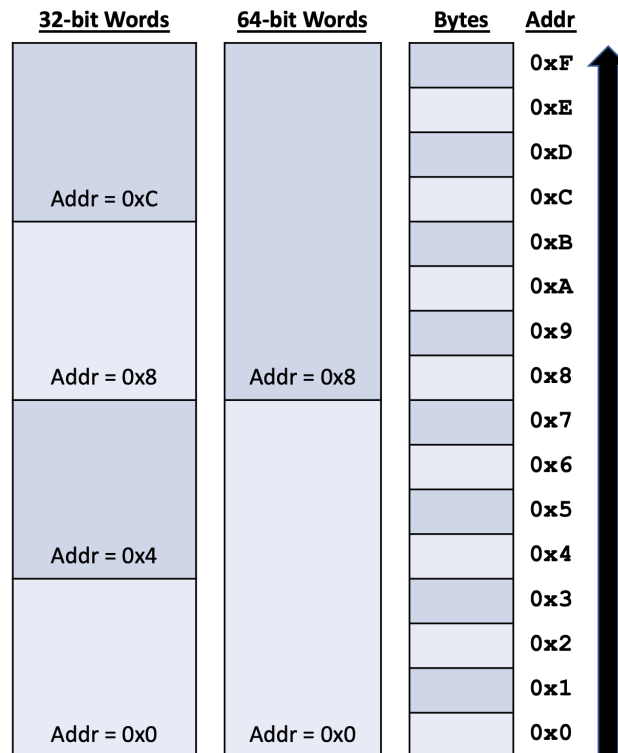
## Addressing

Addresses specify a byte location.

- The address of a word is the address of the first byte in the word
- Addresses of successive words differ by the word size (eg. 32 or 64)

Each column below shows a different view of the same bytes in RAM

- All machines are byte-addressable
- The columns show the word addresses



**Figure 3:** Three different views of RAM.

## Byte Ordering

Any data which spans multiple bytes has a *byte ordering*.

- This refers to the order of the bytes which make up that data.
- This is only for multiple byte data types (so not **chars** or any reference data types)

There are two conventions for byte ordering:

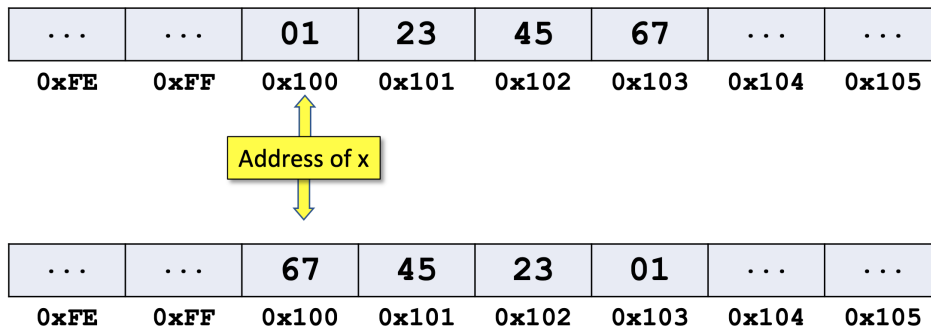
*Big Endian:* The LSB has the highest address. This ordering is used by Sun, PowerPC, and the Internet Byte Ordering

*Little Endian:* The LSB has the lowest address. This ordering is used by Intel and AMD.

*Fun fact:* ARM is a bi-endian system – it can be set to either.

As an example, consider how `int x = 0x1234567;` `&x = 0x100` would look with two different ordering schemes.

Since we take an integer to be 32 bits, we must partition `x` into four buckets of eight bits each. Then `x` would consist of 01 23 45 67.



**Figure 4:** `0x1234567` in both Big Endian (top) and Little Endian (bottom) orderings.

### Question 3

File: `int_endian.c`

As another example, if we run the following on Zeus, what will be printed?

```
int x = 0x1234567;
unsigned char *bp = (unsigned char *) &x;
printf("0x%x\n", *(bp + 2));
```

Since Zeus is Little Endian, the order of our bytes will be reversed. Instead of seeing 01 23 45 67, we'll see 67 45 23 01. Treating it as an array of bytes (which is what `bp` is doing), the second element is 23. As such, we'll see `0x23` printed.

## Examining Data Representations

File: `show_bytes.c`

We can use code to examine the byte representation of data.

- Assign an `unsigned char *` pointer to any data to examine it
  - Remember to cast the data's address to `unsigned char *`

The following code



```
void show_bytes(unsigned char *start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++) {
        printf("%p\t0x%.2x\n", start + i, start[i]);
    }
    printf("\n");
}
```

with `long l = 0xCAFE1337DEAD1701`; prints out

```
0x7ffe8cb58bb0 0x01
0x7ffe8cb58bb1 0x17
0x7ffe8cb58bb2 0xad
0x7ffe8cb58bb3 0xde
0x7ffe8cb58bb4 0x37
0x7ffe8cb58bb5 0x13
0x7ffe8cb58bb6 0xfe
0x7ffe8cb58bb7 0xca
```

## String Byte Ordering

Since each character is a single byte Endian-ness doesn't apply to them.

Given `char s[] = "Hello!"`; `s` will appear in memory as `'H''e''l''l''o''!''\0'`.

## Machine Level Code Representation

How are programs encoded in memory?

- Each program is a sequence of operations
- Each operation is a sequence of bytes

There are two types of program instruction encodings

- Reduced Instruction Set Computer (RISC)
  - Each instruction (operation) is the same number of bytes
  - ARM is an example of a RISC architecture
- Complex Instruction Set Computer (CISC)
  - Each instruction (operation) is a variable number of bytes
  - x86-64 is an example of a CISC architecture


## Representing Instruction in Memory

Let's look at a simple function

```
int sum(int x, int y) {
    return x + y;
}
```

- Alpha and Sun are RISC machines
  - They have two 4-byte instructions
- x86-64 is a CISC machine
  - Seven total instructions
  - The instructions are 1, 2, or 3 bytes in length

		x86	
		C3	
		5D	
		EC	
		89	
		08	
		45	
		03	
		0C	
		45	
		8B	
		E5	
		89	
		55	
DEC Alpha	Sun Sparc		
6B	09		
FA	00		
80	02		
01	90		
42	08		
30	E0		
00	C3		
00	81		



Addr

**Figure 5:** A comparison of an additive function on different architectures.

## Instructions and Endian-ness

- Disassembly
  - Text representation of the binary machine code
  - Generated by a program which reads machine code

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab %ebx
804836c:	83 bb 28 00 00 00 00	\$0x0, 0x28 (%ebx)

- Deciphering values
  - Value: 0x12ab
  - Pad to 32-bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse (Little Endian): ab 12 00 00

## Main Points

- It's all about bits and bytes
  - Number representations
  - Programs (instruction encoding)
  - Text (ASCII/UTF-8 and binary files)
- Different machines follow different conventions
  - Word size (8, 16, 32, or 64 bit)
  - Byte ordering (Little, Big, or Bi-Endian)
  - Data representations

## Boolean Algebra

- Mathematical branch of algebra where expressions result in True or False
  - In computer systems, these values are denoted as
    - \* True: non-zero value
    - \* False: zero value
- There are two types of boolean operations in C: Logical and Bitwise
  - Logical operations only result in True or False
  - Bitwise operations perform boolean operations pairwise, to all the bits in the operands
    - \* The final result is a sequence of bits, each of which is the result of the boolean operation on a pair of corresponding bits

As an example, consider the result of performing `0x3C && 0x95`.

File: `logic.c`

As non-zero integer values, both of them are equivalent to true, so `0x3C && 0x95 == TRUE`.

*Omitted slide 35, Section 2.1.8, definitions of several logical operators.*

Let's revisit our previous example, but instead use a bitwise and: `0x3C & 0x95`.

0	0	1	1	1	1	0	0	0x3C
&	&	&	&	&	&	&	&	
1	0	0	1	0	1	0	1	0x95
↓	↓	↓	↓	↓	↓	↓	↓	
0	0	0	1	0	1	0	0	0x14

**Figure 6:** Result of `0x3C & 0x95`.

*Omitted slide 37, Section 2.1.6, definitions of several bitwise operators.*

## General Boolean Expressions

### Question 4

Given that

$$A = 49_{16} = 1001001_2$$

and

$$B = 55_{16} = 1010101_2$$

what are the results of `A&B`, `A|B`, `A^B`, and `~B`?

Expression	Value
<code>A&amp;B</code>	<code>0x41</code>
<code>A B</code>	<code>0x5D</code>
<code>A^B</code>	<code>0x1C</code>
<code>~B</code>	<code>0xAA</code>

## Bitwise Expressions in C

File: question4.c

Consider the following C snippet:

```
unsigned char A = 0x49;
unsigned char B = 0x55;
printf("0x%.2x\n", A&B);
printf("0x%.2x\n", A|B);
printf("0x%.2x\n", A^B);
printf("0x%.2x\n", (unsigned char)~B);
printf("0x%.2x\n", ~B);
printf("0x%.2x\n", ~B & 0xFF);
```

The above snippet prints out

```
0x41
0x5d
0x1c
0xaa
0xffffffff
0xaa
```

The value of `~B` printed out should give us pause. Why do we need to mask out bits that don't exist in the `char` data type?

Let's double check the data types after performing operations on them.

```
unsigned char A = 0x49;
unsigned char B = 0x55;
printf("  A is %d Bytes\n", sizeof(A));
printf("  B is %d Bytes\n", sizeof(B));
printf(" A&B is %d Bytes\n", sizeof(A&B));
printf(" A|B is %d Bytes\n", sizeof(A|B));
printf(" A^B is %d Bytes\n", sizeof(A^B));
printf(" ~B is %d Bytes\n", sizeof(~B));
printf("A&&B is %d Bytes\n", sizeof(A&&B));
printf("A||B is %d Bytes\n", sizeof(A||B));
printf(" !B is %d Bytes\n", sizeof(!B));
```

When run, this prints out

```
  A is 1 Bytes
  B is 1 Bytes
 A&B is 4 Bytes
 A|B is 4 Bytes
 A^B is 4 Bytes
 ~B is 4 Bytes
A&&B is 4 Bytes
A||B is 4 Bytes
 !B is 4 Bytes
```

---

Everything is an **int** now – this is the result of *integer promotion*, which applies to all arithmetic operations on **chars** and **shorts**.

## Bitwise Masking

File: `mask.c`

### Turning bits on with OR

Suppose that you have a **char** and you want to turn on bit 2 and bit 1 (remember, this is zero-indexed so we're not talking about bit 1 and bit 0). One way we can do this is with a mask and an **OR** operation:

```
value = 0x2A; // 00101010
mask  = 0x06; // 00000110
value |= mask; // 00101110
```

### Turning bits off with AND

Suppose that you have a **char** and you want to turn off bit 2 and bit 1. We can do this with a mask and an **AND** operation.

```
value = 0x2A; // 00101010
mask  = ~0x6; // 11111001
value &= mask; // 00101000
```

### Toggling bits with XOR

Suppose that you have a **char** and you want to toggle (flip) whatever bit 2 and bit 1 are. We can do this with a mask and an **XOR** operation.

```
value = 0x2A; // 00101010
mask  = 0x06; // 00000110
value ^= mask; // 00101100
```

## Shift Operations

C lets you shift bits within data types left (<<) or right (>>).

However, according to the standard (§6.5.7):

The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

The result of  $E1 \ll E2$  is  $E1$  left-shifted  $E2$  bit positions; vacated bits are filled with zeros. If  $E1$  has an unsigned type, the value of the result is  $E1 \times 2^{E2}$ , reduced modulo one more than the maximum value representable in the result type. If  $E1$  has a signed type and nonnegative value, and  $E1 \times 2^{E2}$  is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

The result of  $E1 \gg E2$  is  $E1$  right-shifted  $E2$  bit positions. If  $E1$  has an unsigned type or if  $E1$  has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of  $E1 / 2^{E2}$ . If  $E1$  has a signed type and a negative value, the resulting value is implementation-defined.

### Left Shift

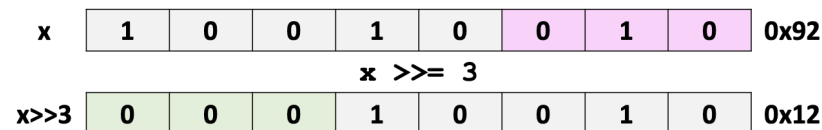


**Figure 7:** An example of C's left shift.

- Shifts the bits in a value  $x$  to the left by  $k$  bits
- Fills in with zeros on the right
- Shifting  $\omega$ -bit value  $x$  left by  $k$  is the same as  $(x \times 2^k) \bmod (2^\omega)$

### Logical Right Shift

- Used with unsigned integer types
- Shifts in zeros from the left
- Shifting  $\omega$ -bit value  $x$  right by  $k$  is the same as  $\lfloor \frac{x}{2^k} \rfloor$



**Figure 8:** An example of C's logical right shift.

