

Representing Letters: ASCII

Representing numbers is great -- but what about words? Can we make sentences with binary data?

- **Key Idea:** Every letter is _____ binary bits.*
(This means that every letter is _____ hex digits.)
- Global standard called the **American Standard Code for Information Interchange (ASCII)** is a _____
_____ for translating numbers to characters.

ASCII Character Encoding Examples:					
Binary	Hex	Char.	Binary	Hex	Char.
0b 0100 0001	0x41	A	0b 0110 0001	0x61	a
0b 0100 0010	0x42	B	0b 0110 0010	0x62	b
		C			c
		D			d
0b0010 0100	0x24	\$	0b0111 1011	0x7b	{

...and now we can form sentences!

Q: Are there going to be any issues with ASCII?

Representing Letters: Other Character Encodings

Since ASCII uses only 8 bits, we are limited to only 256 unique characters. There's far more than 256 characters -- and what about EMOJIs?? 🎉

- **Many** other character encodings exist other than ASCII.
- The most widely used character encoding is known as **Unicode Transformation Format (8-bit)** or _____.
- Standard is **ISO/IEC 10646** (Updated annually!).

Technical Details of UTF-8 Encoding

UTF-8 uses a _____-bit design where each character by be any of the following:

Length	Byte #1	Byte #2	Byte #3	Byte #4
1-byte	0___ ____			
2-bytes	110_ ____	10__ ____		
3-bytes	1110 ____	10__ ____	10__ ____	
4-bytes	1111 0___	10__ ____	10__ ____	10__ ____

Unicode characters are represented by **U+##** (where ## is the hex value of the character encoding data) and all 1-byte characters match the ASCII character encoding:

- 'a' is ASCII _____, or _____.

Example: ε (epsilon) is defined as **U+03b5**. How do we encode this?

Example: I received the following binary message encoded in UTF-8:

0100 1000 0110 1001 1111 0000 1001 1111 1000 1110 1000 1001

1. What is the hexadecimal representation of this message?

2. What is the **byte length** of this message? _____

3. What is the **character length** of this message? _____

4. What does the message say?

02/utf8-binary.c	
4	unsigned char message[] = {
5	0b01001000, 0b01101001, 0b11110000,
	0b10011111, 0b10001110, 0b10001001, 0
6	};
7	printf("%s\n", message);

Bit Manipulation: Binary Addition

For the past two lectures we have focused on the first foundation:

DATA. Today, we are going to begin the transition away from data and into how data applies to the **CPU**. Binary addition work just like decimal addition, but with only **0s** and **1s**:

$$\begin{array}{r} 0b\ 010011 \\ +\ 0b\ \underline{001001} \\ \hline \end{array} \qquad \begin{array}{r} 0b\ 0011 \\ +\ 0b\ \underline{0111} \\ \hline \end{array}$$

Negative Numbers: _____

$$\begin{array}{r} 0b\ 010011 \\ -\ 0b\ \underline{001001} \\ \hline \end{array} \qquad \begin{array}{r} 0b\ 0011 \\ -\ 0b\ \underline{0111} \\ \hline \end{array}$$

Two's Complement

The Two's Complement is a way to represent signed (ex: positive vs. negative) numbers in a way _____!

*For simplicity, let's imagine running on an **7-bit machine**:*

$$-17 =$$

$$-4 =$$

$$-1 =$$

$$\begin{array}{r} 42 \\ -\ \underline{18} \\ \hline \end{array} \qquad \begin{array}{r} 18 \\ -\ \underline{42} \\ \hline \end{array}$$

$$\begin{array}{r} -42 \\ -\ \underline{32} \\ \hline \end{array} \qquad \begin{array}{r} 31 \\ +\ \underline{42} \\ \hline \end{array}$$

Overflow Detection in Two's Complement:

Towards Multiplication

With Two's Complement, we can add and subtract numbers! What about more complex operations?

$$10 \times 2 =$$

$$10 \times 4 =$$

$$10 \times 9 =$$

Bit Shift Operations:

1. [Left Shift]:

2. [Right Shift]: