

CS/COE 0447

Data Representation:
Binary, Hexadecimal,
and Arbitrariness

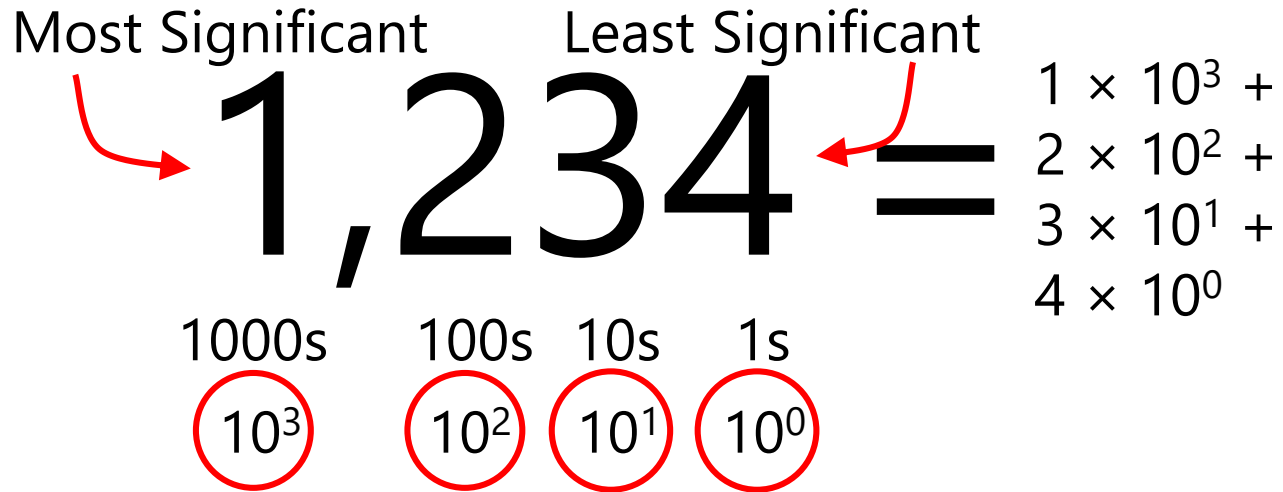
wilkie (with content borrowed from:
Jarrett Billingsley
Dr. Bruce Childers)

Binary

How to convert to and from Base-2 numeral systems

Positional Number Systems

- The numbers we use are written **positionally**: the position of a digit within the number has a meaning.



- How many digit **symbols** do we have in our number system?
 - 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ranges of Representation

- Suppose we have a 4-digit numeric display.
- What is the smallest number it can show?
- What is the biggest number it can show?
- How many *different* numbers can it show?
 - $9999 - 0 + 1 = 10,000$
- What power of 10 is 10,000?
 - 10^4
- **With n digits:**
 - We can **represent** 10^n numbers
 - The **largest number** is $10^n - 1$



Numeric Bases

- These 10s keep popping up... and for good reason
- We use a **base-10 (decimal)** numbering system
 - 10 different digits, and each place is a power of 10
- But we can use (almost) any number as a base!
- The most common bases when dealing with computers are **base-2 (binary)** and **base-16 (hexadecimal)**
- When dealing with multiple bases, you can write the base as a subscript to be explicit about it:

$$5_{10} = 101_2$$

Let's make a base-2 system

- Given base **B**,
 - There are **B** digit symbols
 - Each place is worth **Bⁱ**, starting with **i = 0** on the right
 - Given **n** digits,
 - You can represent **Bⁿ** numbers
 - The largest representable number is **Bⁿ - 1**
- So how about base-2?

Binary (base-2)

- We call a **B**inary dig**IT** a **bit** – a single 1 or 0
- When we say an n -bit number, we mean one with n binary digits

$$\begin{array}{cccccccc} \text{MSB} & & & & & & & \text{LSB} \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 128\text{s} & 64\text{s} & 32\text{s} & 16\text{s} & 8\text{s} & 4\text{s} & 2\text{s} & 1\text{s} \end{array} = \begin{array}{l} 1 \times 128 + \\ 0 \times 64 + \\ 0 \times 32 + \\ 1 \times 16 + \\ 0 \times 8 + \\ 1 \times 4 + \\ 1 \times 2 + \\ 0 \times 1 \\ = 150_{10} \end{array}$$

To convert binary to decimal: ignore 0s, add up place values wherever you see a 1.

Making Change

- You want to give someone \$9.63 in change, using the fewest bills and coins possible. How do you count it out?

$$\text{\$}5 \times \underline{1}$$

$$\text{\$}1 \times \underline{4}$$

$$25\text{¢} \times \underline{2}$$

$$10\text{¢} \times \underline{1}$$

$$5\text{¢} \times \underline{0}$$

$$1\text{¢} \times \underline{3}$$

$$\text{Left: } \$9.63 - \$5 = \$4.63 - \$4 = \$0.63 - 50\text{¢} = \$0.13 - 10\text{¢} = \$0.03 - 0\text{¢} = \$0.03 - 3\text{¢} = \$0.00$$



- Biggest to smallest
- Most significant to least significant
- WHERE COULD THIS BE GOING...

Converting Decimal to Binary

- You want to convert the number 83_{10} to binary.

128s 64s 32s 16s 8s 4s 2s 1s

Left: $83 - 0 = 83$ - $64 = 19$ - $0 = 19$ - $16 = 3$ - $0 = 3$ - $0 = 3$ - $2 = 1$ - $1 = 0$

0 1 0 1 0 0 1 1

- For each place from **MSB**:
- If place value $<$ remainder:
 - digit = 1
 - remainder = remainder - place

$64 < 83$, therefore:

digit = 1

remainder = remainder - 64 ($83 - 64 = 19$)

$32 > 19$, however, so:

digit = 0 (remainder stays the same)

Bits, bytes, nybbles, and words

- A **bit** is one binary digit, and its unit is **lowercase b**.
- A **byte** is an 8-bit value, and its unit is **UPPERCASE B**.
 - This is why your 30 megabit (Mb/s) internet connection can only give you at most 3.75 megabytes (MB) per second!
- A **nybble** (awww!) is 4 bits – half of a byte.
 - Corresponds nicely to a single hex digit.
- A **word** is the "most comfortable size" of number for a CPU.
- When we say "32-bit CPU," we mean its **word** size is 32 bits.
 - This means it can, for example, add two 32-bit numbers at once.
- **BUT WATCH OUT:**
 - Some things (Windows, x86) use **word** to mean **16 bits** and **double word** (or **dword**) to mean **32 bits**.

Why binary? *Whynary?*

- cause it's the easiest thing to implement. :P
 - arithmetic also becomes really easy (as we'll see in several weeks)
 - so, everything on a computer is represented in binary.
-
- **everything.**
 - **EVERYTHING.**
 - **01000101 01010110 01000101 01010010 01011001 01010100 01001000
01001001 01001110 01000111 00101110**
 - (**"EVERYTHING."**)

Hexadecimal

How to convert to and from Base-16 numeral systems

Shortcomings of Binary/Decimal

- Binary numbers can get really long, really quickly.
 - $3,927,664_{10} = 11\ 1011\ 1110\ 1110\ 0111\ 0000_2$
- But nice "round" numbers in binary look arbitrary in decimal.
 - $1000000000000000_2 = 32,768_{10}$
- This is because 10 is not a power of 2!
- We could use base-4, base-8, base-16, base-32, etc.
 - Base-4 is not much terser than binary
 - e.g. $3,927,664_{10} = 120\ 3331\ 2323\ 0000_4$
 - Base-32 would require 32 digit symbols. Yeesh.
 - They do, oddly, have their place... but not really in this context.
 - **Base-8** and **base-16** look promising!

Let's make a base-2 16 system

- Given base **B**,
 - There are **B** digit symbols
 - Each place is worth **Bⁱ**, starting with **i = 0** on the right
 - Given **n** digits,
 - You can represent **Bⁿ** numbers
 - The largest representable number is **Bⁿ - 1**
- So how about base-16?

Hexadecimal or “hex” (base-16)

- Digit symbols after 9 are A-F, meaning 10-15 respectively.
- Usually we call one hexadecimal digit a *hex digit*. No fancy name :(

$$\begin{array}{cccccccc} 0 & 0 & 3 & B & E & E & 7 & 0 \\ 16^7 & 16^6 & 16^5 & 16^4 & 16^3 & 16^2 & 16^1 & 16^0 \end{array} = \begin{array}{l} 0 \times 16^7 + \\ 0 \times 16^6 + \\ 3 \times 16^5 + \\ 11 \times 16^4 + \\ 14 \times 16^3 + \\ 14 \times 16^2 + \\ 7 \times 16^1 + \\ 0 \times 16^0 = \end{array}$$

To convert hex to decimal: use a dang calculator lol

$$3,927,664_{10}$$

Converting from Binary to Hex

- **Four bits** are equivalent to **one hex digit**.
- Converting between them is easy!
- Say we had this binary number:

1110111110111001110000₂

- Starting **from the LSB**, divide into groups of 4 bits (put 0s before the first digits if there are leftovers). Then use the table.

0011 1011 1110 1110 0111 0000

0x3 B E E 7 0

- ↑ (this is common notation for hex, derived from the C language.)

Bin	Hex	Bin	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

know this table.

Binary to Hex

0100 1100 1010 0010 0000 0010 0110 0001

4 C A 2 0 2 6 1

0x4CA20261

32-bits! (Not so bad...)

Base-8??

- base-8, **octal**, used to be commonplace but isn't anymore
- each octal digit (0-7) corresponds to *three* bits
 - this made it a nice fit for 9-, 12-, 18-, and 36-bit machines
- buuuut no one cares about octal anymore ☹️
- SORRY OCTAL
 - it's okay, it has its revenge from time to time
 - try this out in Java sometime: (leading zeroes mean octal, yikes!)

```
System.out.println("012345 = " + 012345);
```

The Powers of Two

- Memorize **at least** the powers up to $\sim 2^8$ or 2^{10} .
 - If you can't remember one, just add the previous one to itself.
- These are the place values for binary, and they are also nice "round" numbers in binary and hex.
- What is the **largest number** that an 8-bit value can hold? What is that in hexadecimal?
 - 255: 0xFF
- How about a 16-bit value?
 - 65535: 0xFFFF
- "0xFFFF" is kinda like "9999" in decimal.
 - What happens if we go beyond that?

	Dec	Hex
2^0	1	0x1
2^1	2	0x2
2^2	4	0x4
2^3	8	0x8
2^4	16	0x10
2^5	32	0x20
2^6	64	0x40
2^7	128	0x80
2^8	256	0x100

Overflow

- In computers, **numbers are finite**.
- Let's say our 4-digit display was counting up: 9997, 9998, 9999...
- What comes "next"?
 - What does this "0000" *really mean*?
 - It **wrapped around**.
- This is **overflow**: the number you are *trying* to represent is **too big to be represented**.
- Essentially, **all arithmetic on the computer is modular arithmetic!**
 - This causes a *lot* of software bugs.



What do we dooo???

- if there's an overflow, that's bad right?
- there are basically three options:
 - **store**
 - **ignore**
 - fall on the **floor** and scream and cry about it
 - (i.e. crash the program)
 - (I* really struggled to come up with a rhyme)
- each of these has pros and cons
- we'll learn more about em later but keep them in your brainmeats

* Jarrett did this, not me

Summing it up (well, soon...)

- everything's in binary, and hex is convenient shorthand
- numbers don't really act like numbers?
- actually, how does the computer know a number is a number?
- how does it know how to add two numbers?
- how does it know how to manipulate strings (for instance, text)?
- how does it know if one pattern of bits is a string or a number or a video or a program or a file or an icon or

IT DOESN'T

Arbitrariness

How binary data is interpreted (or not. I mean who really knows?)

What it means to be “arbitrary”

- it means there's **no reason for it to be that way**.
- we just kinda ***agree*** that it's how things are.
- One of the biggest things I want you to know is:

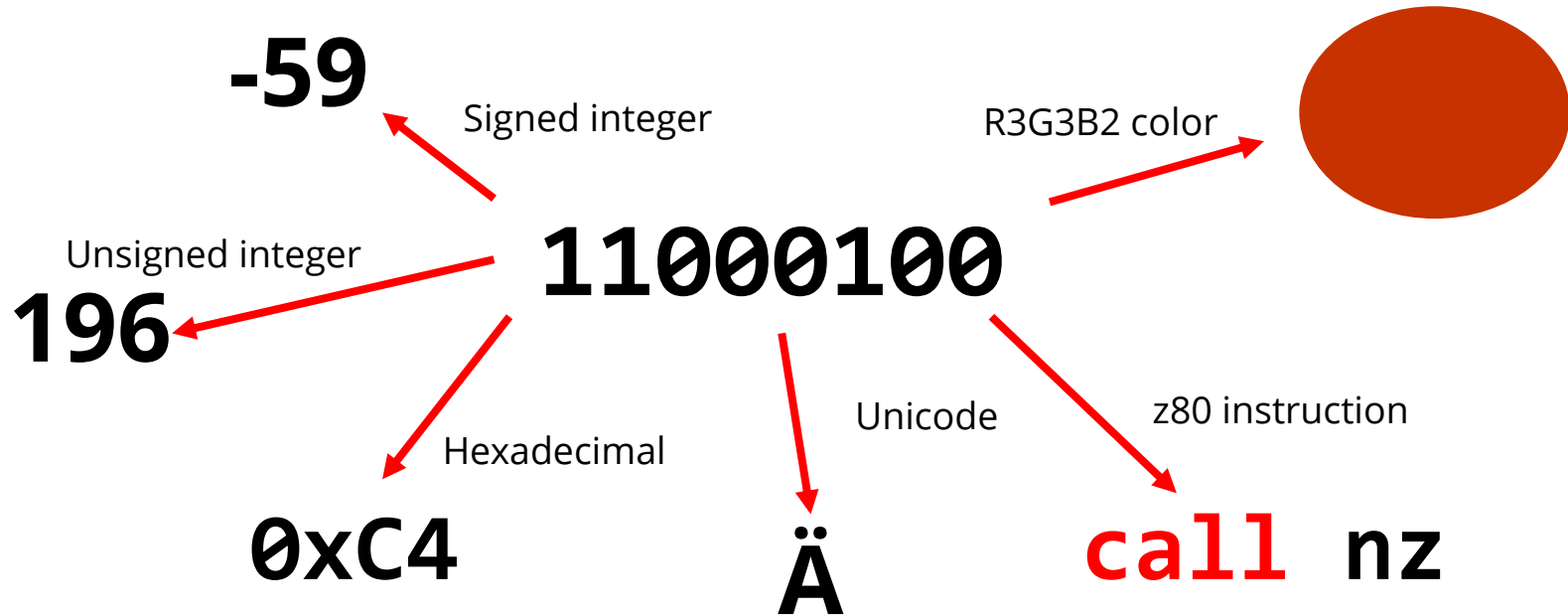
What a pattern of bits *means* is arbitrary.

- As a corollary:

The *same* pattern of bits can be *interpreted* many different ways.

One bit pattern, many meanings

- **All information** on computers is stored as patterns of bits, but...
- How these bits are **interpreted**, **transformed**, and **displayed** is up to the programmer and the user.



The computer doesn't know (or care)

- when writing assembly (and C!) programs, **the computer has no idea what you mean, cause nothing means anything to it**
- "my program assembles/compiles, why is it crashing?"
 - cause the computer is stupid
 - it's a big fast calculator
- there's no difference between nonsense code and useful code
- it's good at doing fun things with bit patterns
- but don't confuse what it does with intelligence
- **every "smart" thing a computer does, it does because a human programmed it to act like that**

Next time...

- Looking at how programs are represented
 - Speaking of arbitrary...
 - We will look at MIPS, tho, specifically
- First taste of Assembly
 - How values are stored, retrieved, manipulated.