# **Data Representation**

Interpreting bits to give them meaning

Part 2: Hexadecimal and Practical Issues

Notes for CSC 100 - The Beauty and Joy of Computing The University of North Carolina at Greensboro

### **Class Reminders**

### Assignments:

• Assignment 1 due in one week (Monday, September 18)

• Review Lab 4 solutions (in Canvas) - important for Lab 5!

### Blown to Bits:

• Chapter 2 discussion - contribute before Wednesday (10:00am)

### From Last Time...

Key points from "Data Representation, Part 1":

- · A number is an abstract idea
- Anything you can point at or write down is a  $\underline{\textit{representation}}$  of a number
- Lots of different representations for the same number:
  - Written in decimal notation (what we're most familiar with)
  - o Written in roman numerals (e.g., 6 is the same as VI) o Written as a set of "tick marks" (e.g., 6 is the same as IIIIII)
  - Written in binary (e.g., 6 is the same as 1102)As a sequence of voltages on wires
- Computers work with binary because switches are off or on (0 or 1)
- Converting between number bases doesn't change the number, just chooses a different representation


### Hexadecimal - another useful base

Hexadecimal is base 16.

How do we get 16 different digits? Use letters!

Hexadecimal digits (or "hex digits" for short): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Counting - now our odometer has 16 digits:

0 <sub>16</sub> (= 0 <sub>10</sub> )	6 <sub>16</sub> (= 6 <sub>10</sub> )	C <sub>16</sub> (= 12 <sub>10</sub> )	12 <sub>16</sub> (= 18 <sub>10</sub> )
1 <sub>16</sub> (= 1 <sub>10</sub> )	7 <sub>16</sub> (= 7 <sub>10</sub> )	D <sub>16</sub> (= 13 <sub>10</sub> )	13 <sub>16</sub> (= 19 <sub>10</sub> )
2 <sub>16</sub> (= 2 <sub>10</sub> )	8 <sub>16</sub> (= 8 <sub>10</sub> )	E <sub>16</sub> (= 14 <sub>10</sub> )	14 <sub>16</sub> (= 20 <sub>10</sub> )
3 <sub>16</sub> (= 3 <sub>10</sub> )	9 <sub>16</sub> (= 9 <sub>10</sub> )	F <sub>16</sub> (= 15 <sub>10</sub> )	15 <sub>16</sub> (= 21 <sub>10</sub> )
4 <sub>16</sub> (= 4 <sub>10</sub> )	$A_{16} (= 10_{10})$	10 <sub>16</sub> (= 16 <sub>10</sub> )	16 <sub>16</sub> (= 22 <sub>10</sub> )
5 <sub>16</sub> (= 5 <sub>10</sub> )	B <sub>16</sub> (= 11 <sub>10</sub> )	11 <sub>16</sub> (= 17 <sub>10</sub> )	17 <sub>16</sub> (= 23 <sub>10</sub> )

### **Hexadecimal/Decimal Conversions**

Conversion process is like binary, but base is 16

Reading digits bottom-up: 423<sub>10</sub> = 1A7<sub>16</sub>

Problem 2: Convert 9C3<sub>16</sub> to decimal: Start with first digit, 9 9\*16 + 12 = 156 156\*16 + 3 = 2499

Therefore, 9C3<sub>16</sub> = 2499<sub>10</sub>

Hex Digit List

 $\begin{array}{c} 0 = 0 \\ 1_{16} = 1_{10} \\ 2_{16} = 2_{10} \\ 3_{16} = 2_{10} \\ 3_{16} = 4_{10} \\ 5_{16} = 5_{10} \\ 5_{16} = 5_{10} \\ 6_{16} = 6_{10} \\ 8_{16} = 8_{10} \\ 9_{16} = 9_{10} \\ 8_{16} = 11_{10} \\ C_{16} = 12_{10} \\ D_{16} = 14_{10} \\ F_{16} = 15_{10} \end{array}$ 

### **Hexadecimal/Decimal Conversions**

Conversion process is like binary, but base is 16

<u>Problem 1</u>: Convert 423 $_{10}$  to hexadecimal: 423/16 = quotient 26, remainder 7 (=7 $_{16}$ ) 26/16 = quotient 1, remainder 10 (=A<sub>16</sub>) 1/16 = quotient 0, remainder 1 (=1<sub>16</sub>)

Reading digits bottom-up: 423<sub>10</sub> = 1A7<sub>16</sub>

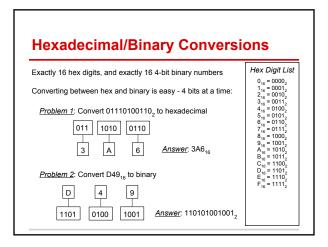
<u>Problem 2</u>: Convert 9C3<sub>16</sub> to decimal: Start with first digit, 9 9\*16 + 12 = 156 156\*16 + 3 = 2499

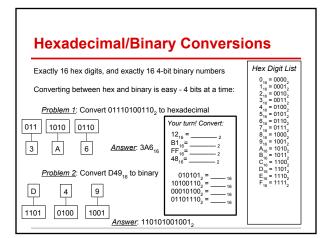
Therefore, 9C3<sub>16</sub> = 2499<sub>10</sub>

 $\begin{array}{c} 0_{16} = 0_{10} \\ 1_{16} = 1_{10} \\ 2_{16} = 2_{10} \\ 3_{16} = 3_{10} \\ 3_{16} = 6_{10} \\ 5_{16} = 6_{10} \\ 7_{16} = 6_{10} \\ 7_{16} = 8_{10} \\ 9_{16} = 8_{10} \\ 9_{16} = 10_{10} \\ C_{16} = 12_{10} \\ C_{16} = 13_{10} \\ E_{16} = 14_{10} \\ E_{16} = 15_{10} \\ E_{16} = 15_{10} \\ \end{array}$ 

Hex Digit List

Your turn! Convert: 103<sub>10</sub> = 247<sub>10</sub> = 95210 = \_ **—** 16 3C<sub>16</sub> = \_\_ B9<sub>16</sub> = \_\_ 357<sub>16</sub> = \_\_ - 10





# 

D	
Remember	
	-
Don't get lost in the details and manipulations:	
Any base is a representation of an abstract number	
We are interested in working with the number, and	
computations are not "in a base" - the base is only useful for having it make sense to us or the computer	
for having it make sense to as or the compater	
	1
Practice!	
You should be able to convert from one base to another.	
Lots of ways to practice:  • By hand: Pick a random number convert to binary and convert	
back - did you get the same value?  o This isn't foolproof: You could have made two mistakes!	
With a calculator: Many calculators (physical and software) do base conversion - check your randomly selected conversions.	
With a web site: Several web sites provide says to practice	
For example, see <a href="http://cs.jupui.edu/~aharris/230/binPractice.html">http://cs.jupui.edu/~aharris/230/binPractice.html</a>	
	1
Practical Issues with Numbers	
Finite Length Integers	
Question (a little contrived):	
If a CPU has 4 single-bit storage locations for each number, what happens when you add:	
1111 <sub>2</sub> + 0001 <sub>2</sub> =	
-	

# Practical Issues with Numbers Finite Length Integers Question (a little contrived): If a CPU has 4 single-bit storage locations for each number, what happens when you add:

1111<sub>2</sub> + 0001<sub>2</sub> = \_\_\_\_\_\_<sub>2</sub>

Answer Part 1: If you did this on paper, you'd get 10000,

Which leads to another question:

How do we store 5 bits when there are only storage locations for 4 bits?

### **Practical Issues with Numbers**

Finite Length Integers

Question (a little contrived):

Answer Part 1: If you did this on paper, you'd get 10000,

Which leads to another question:

How do we store 5 bits when there are only storage locations for 4 bits?

<u>Answer Part 2</u>: What CPUs do is throw out the 5th bit, storing  $0000_2$  Which means: To a 4-bit computer, 15 + 1 = 0

### **Practical Issues with Numbers**

Finite Length Integers

On real computers:

- This happens, but with 32-bit numbers or 64-bit numbers instead of 4.
- When things "wrap around" it actually goes to negative values...
   On a 32-bit CPU: 2,147,483,647 + 1 = -2,147,483,648

However: Some programming languages/systems support numbers larger than the hardware, by using multiple memory locations.

Let's try this!

## **Practical Issues with Numbers** Finite Length Integers x = 1000\*1000\*1000\*1000 print x Outputs: Outputs: Outputs: -727379968 1000000000000 -727379968 **Practical Issues with Numbers** Finite Length Integers In C: int val=1000\*1000\*1000\*1000; printf("%d\n", val); x = 1000\*1000\*1000\*1000 print x Outputs: Outputs: Outputs: -727379968 -727379968 1000000000000 First thought: Python is cool! Second thought: Don't expect something for nothing... Let's do something pretty useless (that takes a lot of integer operations) Problem: Compute the last 6 digits of the billionth Fibonacci number **Practical Issues with Numbers** Finite Length Integers In C: In Python: int val=1000\*1000\*1000\*1000; printf("%d\n", val); int val = 1000\*1000\*1000\*1000; System.out.println(val); x = 1000\*1000\*1000\*1000 print x Outputs: Outputs: Outputs: -727379968 10000000000000 -727379968 First thought: Python is cool! Second thought: Don't expect something for nothing... Let's do something pretty useless (that takes a lot of integer operations) Problem: Compute the last 6 digits of the billionth Fibonacci number In C: In Java: In Python: 3.5 seconds 3.4 seconds 3 minutes, 56.2 seconds Times on my laptop: Intel i7-3740QM (2.7GHz)

# **Practical Issues with Numbers** Finite Precision Floating Point Question: How do you write out 1/3 in decimal? Answer: 0.33333333333.... Observation: Impossible to write out exactly with a finite number of digits The same holds in binary! Can be written exactly Cannot be written exactly 1/3 = 0.0101010101...<sub>2</sub> 0.5 = 0.1<sub>2</sub> 0.25 = 0.01<sub>2</sub> % = 0.001100110011. 1/10 = 0.0001100110011..., 0.375 = 0.011, Imagine: How hard is it to write banking software when there is no finite representation of a dime (0.10 dollars)?!?!? Solutions people came up with: Work with cents (integers!) or special codings (BCD=Binary Coded Decimal) **Practical Issues with Numbers** Finite Precision Floating Point Question: How do you write out 1/3 in decimal? Bottom Line: Observation: There are a lot of subtle problems with numbers that go beyond the level of study in CSC 100 Can be These issues usually don't come up. 0.5 = 0 0.25 = But... when they matter, they can matter a LOT. 0.375 For now: Be aware what the issues are. Imagine: Hov For a later class: Understand the details. Solutions people came up with: Work with cents (integers!) or special codings (BCD=Binary Coded Decimal) **Still More Data Representation for Later** Now we know all about representing numbers But computers also deal with text, web pages, pictures, sound/music, video, ... How does that work?