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)

Lab Exercises:

- 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 *representation* of a number
- Lots of different representations for the same number:
  - Written in decimal notation (what we’re most familiar with)
  - Written in roman numerals (e.g., 6 is the same as VI)
  - 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 110_2)
  - 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_{16} (= 0_{10})	6_{16} (= 6_{10})	C_{16} (= 12_{10})	12_{16} (= 18_{10})	...
1_{16} (= 1_{10})	7_{16} (= 7_{10})	D_{16} (= 13_{10})	13_{16} (= 19_{10})	
2_{16} (= 2_{10})	8_{16} (= 8_{10})	E_{16} (= 14_{10})	14_{16} (= 20_{10})	
3_{16} (= 3_{10})	9_{16} (= 9_{10})	F_{16} (= 15_{10})	15_{16} (= 21_{10})	
4_{16} (= 4_{10})	A_{16} (= 10_{10})	10_{16} (= 16_{10})	16_{16} (= 22_{10})	
5_{16} (= 5_{10})	B_{16} (= 11_{10})	11_{16} (= 17_{10})	17_{16} (= 23_{10})	

Hexadecimal/Decimal Conversions

Conversion process is like binary, but base is 16

Problem 1: Convert 423_{10} to hexadecimal:

423/16 = quotient 26, remainder 7 (=7_{16})

26/16 = quotient 1, remainder 10 (=A_{16})

1/16 = quotient 0, remainder 1 (=1_{16})

- Reading digits bottom-up: 423_{10} = $1A7_{16}$

Problem 2: Convert $9C3_{16}$ to decimal:

Start with first digit, 9

9*16 + 12 = 156

156*16 + 3 = 2499

- Therefore, $9C3_{16}$ = 2499_{10}

Hex Digit List

$0_{16} = 0_{10}$
$1_{16} = 1_{10}$
$2_{16} = 2_{10}$
$3_{16} = 3_{10}$
$4_{16} = 4_{10}$
$5_{16} = 5_{10}$
$6_{16} = 6_{10}$
$7_{16} = 7_{10}$
$8_{16} = 8_{10}$
$9_{16} = 9_{10}$
$A_{16} = 10_{10}$
$B_{16} = 11_{10}$
$C_{16} = 12_{10}$
$D_{16} = 13_{10}$
$E_{16} = 14_{10}$
$F_{16} = 15_{10}$

Hexadecimal/Decimal Conversions

Conversion process is like binary, but base is 16

Problem 1: Convert 423_{10} to hexadecimal:

423/16 = quotient 26, remainder 7 (=7_{16})

26/16 = quotient 1, remainder 10 (=A_{16})

1/16 = quotient 0, remainder 1 (=1_{16})

- Reading digits bottom-up: 423_{10} = $1A7_{16}$

Problem 2: Convert $9C3_{16}$ to decimal:

Start with first digit, 9

9*16 + 12 = 156

156*16 + 3 = 2499

- Therefore, $9C3_{16}$ = 2499_{10}

Your turn! Convert:

103_{10} = ______ $_{16}$
247_{10} = ______ $_{16}$
952_{10} = ______ $_{16}$

$3C_{16}$ = ______ $_{10}$
$B9_{16}$ = ______ $_{10}$
357_{16} = ______ $_{10}$

Hex Digit List

$0_{16} = 0_{10}$
$1_{16} = 1_{10}$
$2_{16} = 2_{10}$
$3_{16} = 3_{10}$
$4_{16} = 4_{10}$
$5_{16} = 5_{10}$
$6_{16} = 6_{10}$
$7_{16} = 7_{10}$
$8_{16} = 8_{10}$
$9_{16} = 9_{10}$
$A_{16} = 10_{10}$
$B_{16} = 11_{10}$
$C_{16} = 12_{10}$
$D_{16} = 13_{10}$
$E_{16} = 14_{10}$
$F_{16} = 15_{10}$

Hexadecimal/Binary Conversions

Exactly 16 hex digits, and exactly 16 4-bit binary numbers

Converting between hex and binary is easy - 4 bits at a time:

Problem 1: Convert 01110100110_2 to hexadecimal

011 | 1010 | 0110

3 | A | 6

Answer: $3A6_{16}$

Problem 2: Convert $D49_{16}$ to binary

D | 4 | 9

1101 | 0100 | 1001

Answer: 110101001001_2

Hex Digit List

$0_{16} = 0000_2$
$1_{16} = 0001_2$
$2_{16} = 0010_2$
$3_{16} = 0011_2$
$4_{16} = 0100_2$
$5_{16} = 0101_2$
$6_{16} = 0110_2$
$7_{16} = 0111_2$
$8_{16} = 1000_2$
$9_{16} = 1001_2$
$A_{16} = 1010_2$
$B_{16} = 1011_2$
$C_{16} = 1100_2$
$D_{16} = 1101_2$
$E_{16} = 1110_2$
$F_{16} = 1111_2$

Hexadecimal/Binary Conversions

Exactly 16 hex digits, and exactly 16 4-bit binary numbers

Converting between hex and binary is easy - 4 bits at a time:

Problem 1: Convert 01110100110_2 to hexadecimal

011 | 1010 | 0110

3 | A | 6

Answer: $3A6_{16}$

Problem 2: Convert $D49_{16}$ to binary

D | 4 | 9

1101 | 0100 | 1001

Answer: 110101001001_2

Your turn! Convert:

$12_{16} =$ ________$_2$
$B1_{16} =$ ________$_2$
$FF_{16} =$ ________$_2$
$48_{16} =$ ________$_2$

$010101_2 =$ _____$_{16}$
$10100110_2 =$ _____$_{16}$
$00010100_2 =$ _____$_{16}$
$01101110_2 =$ _____$_{16}$

Hex Digit List

$0_{16} = 0000_2$
$1_{16} = 0001_2$
$2_{16} = 0010_2$
$3_{16} = 0011_2$
$4_{16} = 0100_2$
$5_{16} = 0101_2$
$6_{16} = 0110_2$
$7_{16} = 0111_2$
$8_{16} = 1000_2$
$9_{16} = 1001_2$
$A_{16} = 1010_2$
$B_{16} = 1011_2$
$C_{16} = 1100_2$
$D_{16} = 1101_2$
$E_{16} = 1110_2$
$F_{16} = 1111_2$

Use of hexadecimal in file dumps

Binary is a very long format (8 bits per byte), but often data files only make sense as binary data. Hexadecimal is great for this - simple one-to-one correspondence with binary, and more compact.

Sample "file dump":

```
0000000: ffd8 ffe1 35fe 4578 6966 0000 4949 2a00  ....5.Exif..II*.
0000010: 0800 0000 0b00 0e01 0200 2000 0000 9200  .......... .....
0000020: 0000 0f01 0200 0600 0000 b200 0000 1001  ................
0000030: 0200 1900 0000 b800 0000 1201 0300 0100  ................
0000040: 0000 0600 0000 1a01 0500 0100 0000 d800  ................
0000050: 0000 1b01 0500 0100 0000 e000 0000 2801  ..............(.
0000060: 0300 0100 0000 0200 0000 3201 0200 1400  ..........2.....
0000070: 0000 e800 0000 1302 0300 0100 0000 0200  ................
0000080: 0000 6987 0400 0100 0000 fc00 0000 2588  ..i...........%.
0000090: 0400 0100 0000 2413 0000 f213 0000 2020  ......$.......
00000a0: 2020 2020 2020 2020 2020 2020 2020 2020
00000b0: 2020 2020 2020 2020 2020 2020 2000 4361                .Ca
00000c0: 6e6f 6e00 4361 6e6f 6e20 506f 7765 7253  non.Canon PowerS
00000d0: 686f 7420 5358 3233 3020 4853 0000 0000  hot SX230 HS....
00000e0: 0000 0000 b400 0000 0100 0000 b400 0000  ................
00000f0: 0100 0000 3230 3131 3a30 373a 3134 2031  ....2011:07:14 1
0000100: 353a 3039 3a32 3700 2100 9a82 0500 0100  5:09:27.!.......
0000110: 0000 8e02 0000 9d82 0500 0100 0000 9602  ................
0000120: 0000 2788 0300 0100 0000 6400 0000 3088  ..'.......d...0.
```

Position in file | Actual binary data (written in hexadecimal) | The same data, showing character representation

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

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?
  - 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 http://cs.iupui.edu/~aharris/230/binPractice.html

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:

$$1111_2 + 0001_2 = \_\_\_\_\_\__2$$

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:

$1111_2 + 0001_2 = \_\_\_\_\_\__2$

Answer Part 1: If you did this on paper, you'd get 10000_2

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):

If a CPU has 4 single-bit storage locations for each number, what happens when you add:

$1111_2 + 0001_2 = \_\_\_\_\_\__2$

Answer Part 1: If you did this on paper, you'd get 10000_2

Which leads to another question:

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

Answer Part 2: 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

In C:

```
int val=1000*1000*1000*1000;
printf("%d\n", val);
```

Outputs:

```
-727379968
```

In Java:

```
int val = 1000*1000*1000*1000;
System.out.println(val);
```

Outputs:

```
-727379968
```

In Python:

```
x = 1000*1000*1000*1000
print x
```

Outputs:

```
1000000000000
```


Practical Issues with Numbers

Finite Length Integers

In C:

```
int val=1000*1000*1000*1000;
printf("%d\n", val);
```

Outputs:

```
-727379968
```

In Java:

```
int val = 1000*1000*1000*1000;
System.out.println(val);
```

Outputs:

```
-727379968
```

In Python:

```
x = 1000*1000*1000*1000
print x
```

Outputs:

```
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:

```
int val=1000*1000*1000*1000;
printf("%d\n", val);
```

Outputs:

```
-727379968
```

In Java:

```
int val = 1000*1000*1000*1000;
System.out.println(val);
```

Outputs:

```
-727379968
```

In Python:

```
x = 1000*1000*1000*1000
print x
```

Outputs:

```
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

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 ⅓ 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
$0.5 = 0.1_2$ $0.25 = 0.01_2$ $0.375 = 0.011_2$	$⅓ = 0.0101010101..._2$ $⅕ = 0.001100110011..._2$ $1/10 = 0.0001100110011..._2$

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 ⅓ in decimal?

Observation:

Can be
0.5 = 0
0.25 =
0.375 =

Bottom Line:

There are a lot of subtle problems with numbers that go beyond the level of study in CSC 100

These issues *usually* don't come up.

But… when they matter, they can matter a LOT.

For now: Be aware what the issues are.

For a later class: Understand the details.

Imagine: How
representatio

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?