Data Representation

2

CS/COE 0449
Introduction to Systems Software

wilkie
(with content borrowed from Vinicius Petrucci and Jarrett Billingsley)

Spring 2019/2020

Binary Encoding

Bits, Bytes, and Nybbles

Positional Number Systems

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

Most Significant Least Significant

$$1{,}234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1000s	100s	10s	1s
10^3	10^2	10^1	10^0

- 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 $\mathbf{B^i}$, starting with $\mathbf{i = 0}$ on the right
  - Given **n** digits,
    - You can represent $\mathbf{B^n}$ numbers
    - The largest representable number is $\mathbf{B^n - 1}$
- **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

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

$1 \times 128 +$
$0 \times 64 +$
$0 \times 32 +$
$1 \times 16 +$
$0 \times 8 +$
$1 \times 4 +$
$1 \times 2 +$
0×1

$= 150_{10}$

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

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 mega*bit* (Mb/s) internet connection can only give you at most 3.75 mega*bytes* (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.**

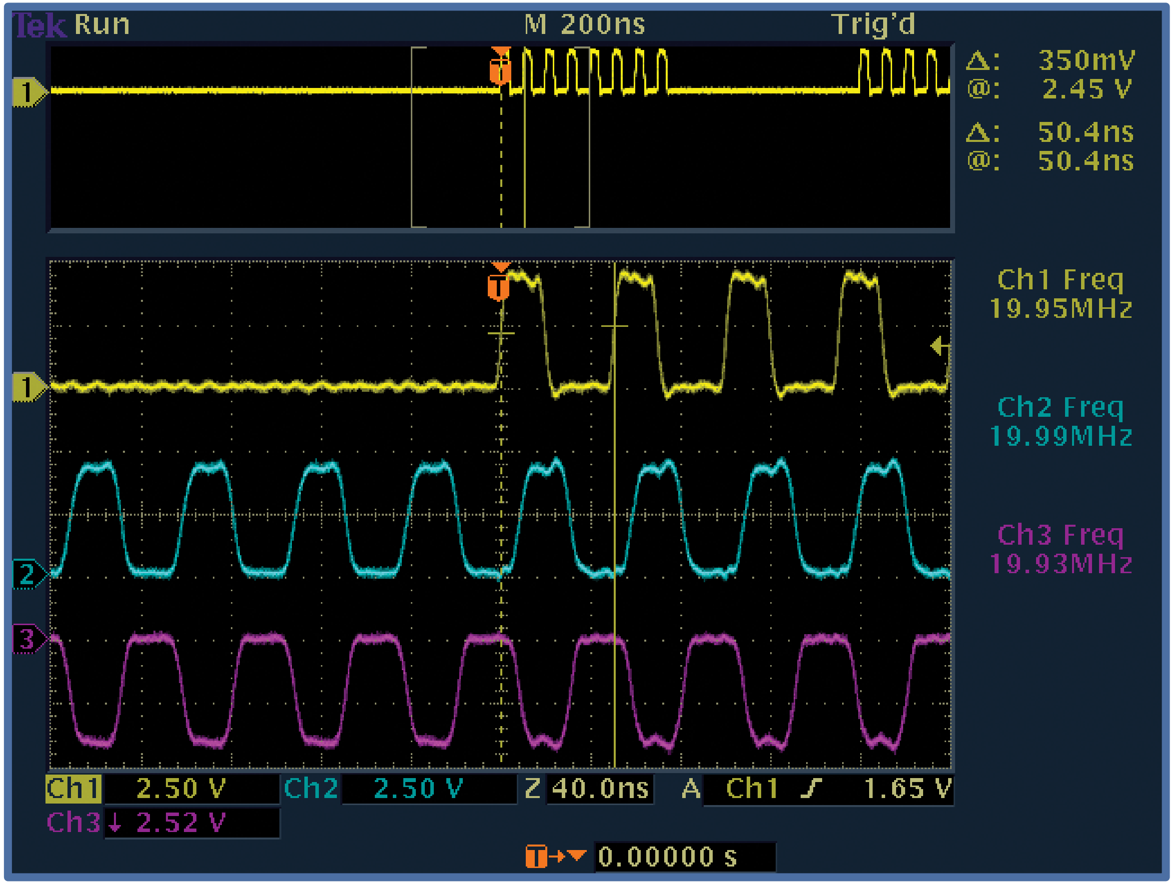
Why binary? *Whynary?*

- Because it's the easiest thing to implement!
- Basic arithmetic is a bit easier.
- So, everything on a computer is represented in binary.

  - **everything.**
  - ***EVERYTHING.***
  - ***01000101 01010110 01000101 01010010 01011001 01010100 01001000 01001001 01001110 01000111 00101110***
    - *(“EVERYTHING.”)*

Binary Representation

- Computers translate electrical signals to either 0 or 1.
- It is relatively easy to devise electronics that operate this way.
- In reality, there is no such thing as “binary” so we often have to approximate and mitigate error.



Oscilloscope visualization of several digital wires. From @computerfact on Twitter.

Integer Encoding

Casting is Not Just a Witch or Wizard Thing

Hexadecimal

- Binary numbers can get really long, 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!

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

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 :(

$$\underset{16^7}{0}\ \underset{16^6}{0}\ \underset{16^5}{3}\ \underset{16^4}{B}\ \ \underset{16^3}{E}\ \underset{16^2}{E}\ \underset{16^1}{7}\ \underset{16^0}{0} =$$

$$\begin{aligned} 0 \times 16^7 &+ \\ 0 \times 16^6 &+ \\ 3 \times 16^5 &+ \\ 11 \times 16^4 &+ \\ 14 \times 16^3 &+ \\ 14 \times 16^2 &+ \\ 7 \times 16^1 &+ \\ 0 \times 16^0 &= \end{aligned}$$

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

To convert hex to decimal: use a dang calculator lol

Binary to Hex

(animated)

0100 1100 1010 0010 0000 0010 0110 0001

4 C A 2 0 2 6 1

0x4CA20261

32-bits! (Not so bad…)

Q: Create a random binary string and practice!

Signed Numbers (sign-magnitude)

- Seems like a good time to think about "negative" values.
  - These are numbers that have nothing good to say.
- Binary numbers have bits which are either 0 or 1.
  - Well, yeah…
- So what if we used one bit to designate "positive" or "negative"
  - Called **sign-magnitude** encoding:

$$\mathtt{1}\underbrace{\mathtt{0100010}} = -\underbrace{34}$$

$$\mathtt{0}\underbrace{\mathtt{0010110}} = \underbrace{22}\ \text{(normal)}$$

Signed Numbers (problems)

$$1\underbrace{0000000} = -\underbrace{0}$$

$$0\underbrace{0000000} = \underbrace{0}$$

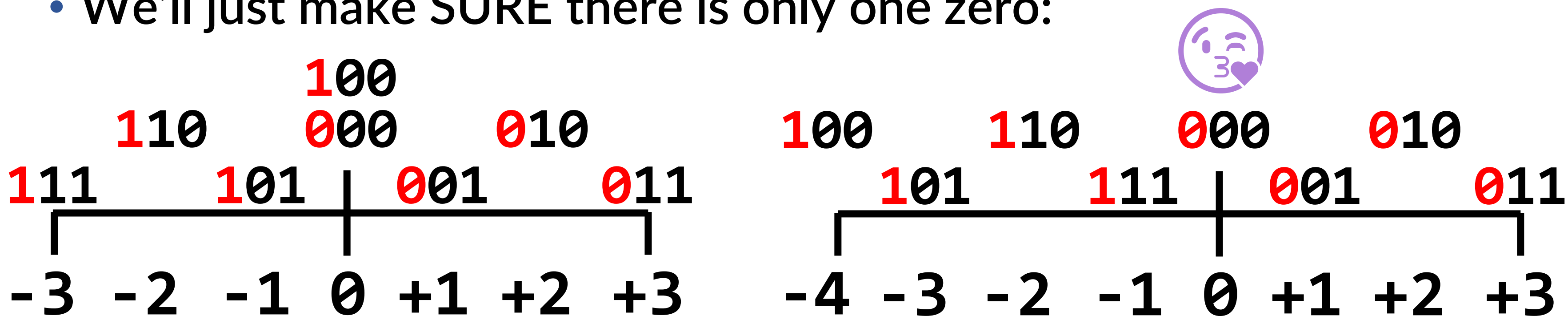
- **Waaaaait a second.**
  - What is negative zero???

- **This encoding allows two different zeros.**
  - This means we can represent how many different values (8-bit)?
    - $2^8 - 1$ (minus the one redundant value) = 255 (-127 … 0 … 127)

- **Sign-magnitude is a little naïve… let's try a different approach…**

Signed Numbers (2's Complement)

- This one, I promise, is juuuuust right.
  - But it's a little strange!
- We'll just make SURE there is only one zero:



Signed Magnitude

2's Complement

- So, we flip the bits… and add one.
  - Adding one makes sure our -0 is used for -1 instead!
- Sure, it's a little lopsided, but, hey, we get an extra number.
  - But, hmm, but -4 **doesn't have a valid positive number**.
    - That's the trade-off, but it's for the best.

Signed Numbers (2's Complement)

- Let's look some examples:

11010100=-00101011=-(43+1)=-44

00100110= 00100110= 38

00000000= 00000000= 0

11111111=-00000000=-(0+1) =-1

- If the MSB is 1: Flip! Add one!
- Otherwise: Do nothing! It's the same!

Signed Numbers (2's Complement)

- What happens when we add zeros to a positive number:

00100110 = 38

000

10100110 = ?

-(01011001+1) = ?

-01011010 = -90

- What happe :

10100110 = -90

1111111110100110 =

-0000000001011001 = -90

Dang that's cool!

Can I Get an Extension?

- Sometimes you need to *widen* a number with fewer bits to more
- ***zero extension*** is easy: put 0s at the beginning.

1001_2 ➔ *to 8 bits* ➔ $0000\ 1001_2$

- But there are also signed numbers… what about those?
  - The **top bit (MSB)** of signed numbers determines **the sign (+/-)**
- ***sign extension*** puts *copies of the sign bit* at the beginning

1001_2 ➔ *to 8 bits* ➔ $1111\ 1001_2$

0010_2 ➔ *to 8 bits* ➔ $0000\ 0010_2$

Q: What happens when you sign extend the largest unsigned value?

Absolutely Bonkers

```
public class AbsTest {
  public static int abs(int x) {
    if (x < 0) {
      x = -x;
    }

    return x;
  }

  public static void main(String[] args) {
    System.out.println(
      String.format("|%d| = %d", Integer.MIN_VALUE, AbsTest.abs(Integer.MIN_VALUE))
    );
  }
}                                         // Outputs: |-2147483648| = -2147483648
```

Q: How many bits is a Java `int`? What happened here?

Integer Ranges

- **Recall:**
  - The range of an unsigned integer is 0 to $2^n - 1$
  - **Q**: Why do we subtract 1?

- **What is the range of a 2's complement number?**
  - Consider the sign bit, how many negative integers?
  - Consider, now, the positive integers.
  - Remember 0.

$$-2^{n-1} \text{ to } 2^{n-1} - 1$$

Q: What if you needed a range with far more negatives than positives?

Integers in C

- **C allows for variables to be declared as either signed or unsigned.**
  - Remember: “signed” does not mean “negative” just that it *can* be negative.

- **An unsigned integer variable has a range from 0 to $2^n – 1$**
- **And signed integers are usually 2’s complement: 2^{n-1} to $2^{n-1} – 1$**
  - Where “n” is determined by the variable’s size in bits.

- **Integer Types: (signed by default, their sizes are arbitrary!!)**
  - `char` `unsigned char` 8 bits (byte)
  - `short int` `unsigned short int` 16 bits (half-word)
  - `int` `unsigned int` 32 bits (word)
  - `long int` `unsigned long int` 64 bits (double-word)

- **Usually no strong reason to use anything other than (un)signed int.**

Q: What is the range of a `signed char`?

Integers in C: Limits

- **Since sizes of integers are technically arbitrary…**
  - They are usually based on the underlying architecture.
- **… C provides standard library constants defining the ranges.**
  - https://pubs.opengroup.org/onlinepubs/009695399/basedefs/limits.h.html

```c
#include <limits.h>            // Provides INT_MAX etc
#include <stdio.h>             // Provides printf

int main() {
  printf("%d ",  INT_MAX);     // Print the maximum signed int
  printf("%u\n", UINT_MAX);    // Print the maximum unsigned int
  return 0;
}               // Output: 2147483647 4294967295
```

Q: Experiment with using "%d" for both. What is the result?

Casting

- **C lets you move a value from an unsigned integer variable to a signed integer variable. (and vice versa)**

- **However, this is not always valid! Yet, it will do it anyway.**
  - The binary value is the same, ***its interpretation is not***!
    - This is called ***coercion***, and this is a relatively simple case of it.
  - Since it ignores obvious invalid operations this is sometimes referred to as “weak” typing.
  - The strong/weak terminology has had very fragile definitions over the years and are arguably useless in our context. Let’s ignore them.

- **Moving values between different types is called *casting***
  - Which sounds magical and it sometimes is.

Q: What is true of the result of casting the value -1 to an unsigned type?