Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Floating-Point Numbers, Character Codes, Error Detection

Homework #1 Review

- Solutions will be posted in Sakai (resources folder)
  - *Why Sakai? Only available to class members…*
- **Review 50-word "essay"**
- **Review 2.7(a,b)**
  - Represent **60** in 8-bit
    - Signed magnitude?
    - One's complement?
    - Two's complement?
  - Represent **-60** in 8-bit
    - Signed magnitude?
    - One's complement?
    - Two's complement?

Upcoming Events

- **Homework #2**
  - Assigned today
  - Due **Thursday**
  - 3 problems
- **Quiz #1**
  - **Thursday**
  - Topics?
    - Material from Homework #1 and #2
    - Material from Intro lecture

Why Floating-Point?

- Existing representations deal with integer values only
  - Signed magnitude
  - One's complement
  - Two's complement
- Adding in a fixed decimal point is awkward / inflexible
- Scientific and business applications need a standardized way to deal with real number values
  - **Floating-point numbers**

Floating-Point Representation

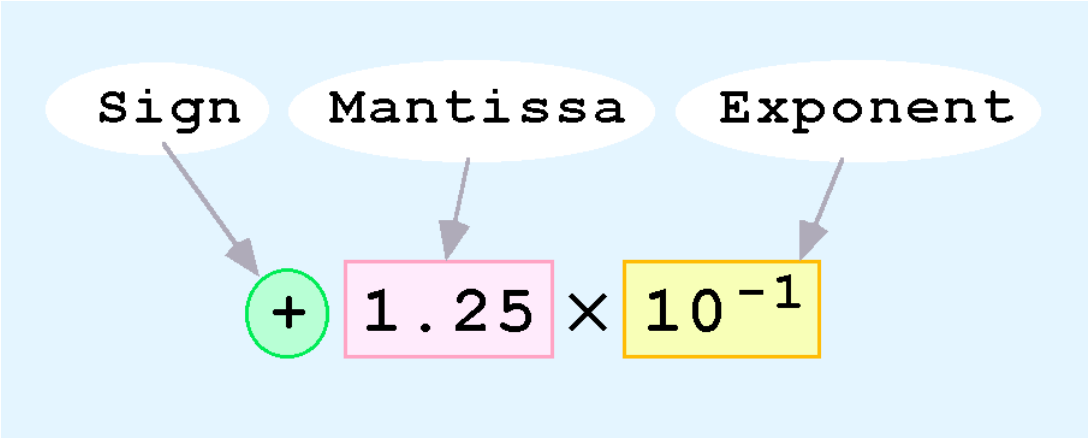
- Do we need hardware or software?
  - Clever programmers can do floating-point purely in software
  - Drawbacks: Complicated, slow
- Modern computers have **specialized hardware** that directly performs floating-point arithmetic

Floating-Point Representation

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in **scientific notation**
  - For example:
    - $0.125 = 1.25 \times 10^{-1}$
    - $5{,}000{,}000 = 5.0 \times 10^{6}$

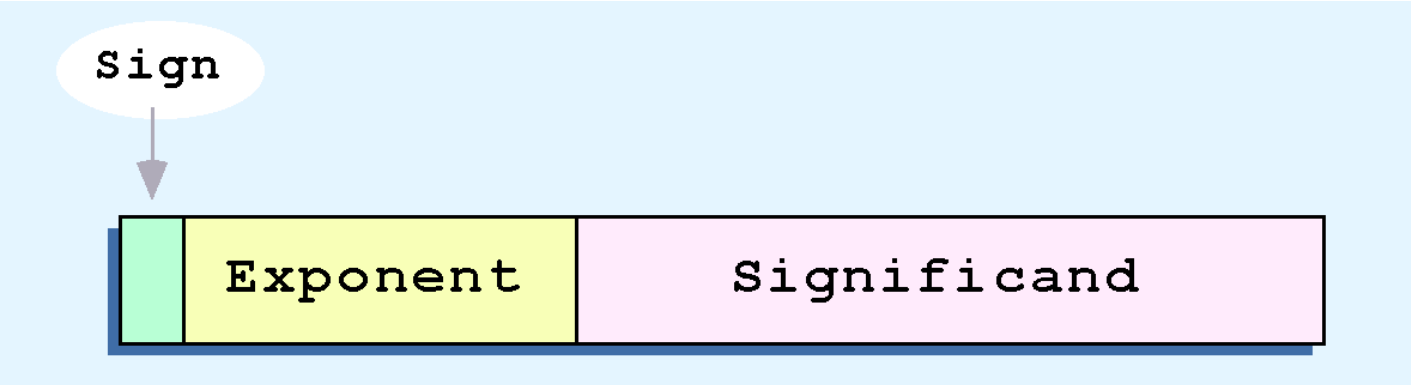
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

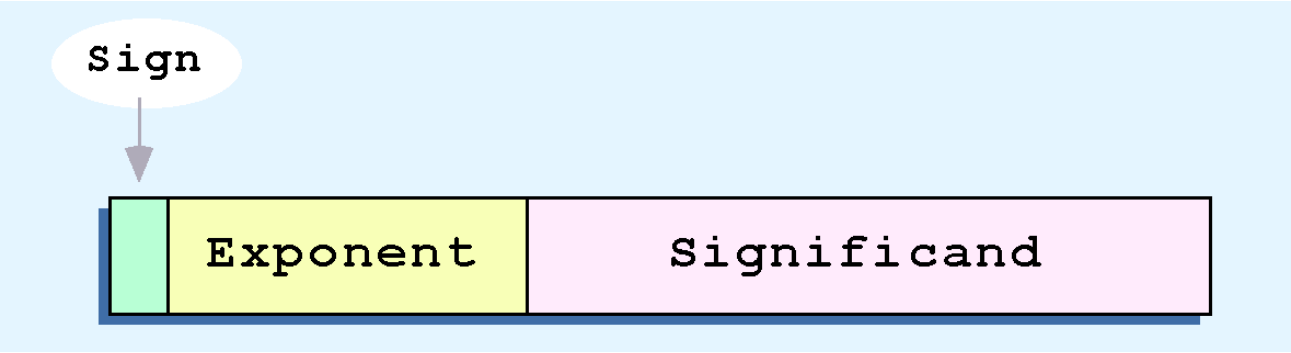
- Computer representation of a floating-point number consists of three fixed-size fields:
- This is the standard arrangement of these fields:



Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

Floating-Point Representation

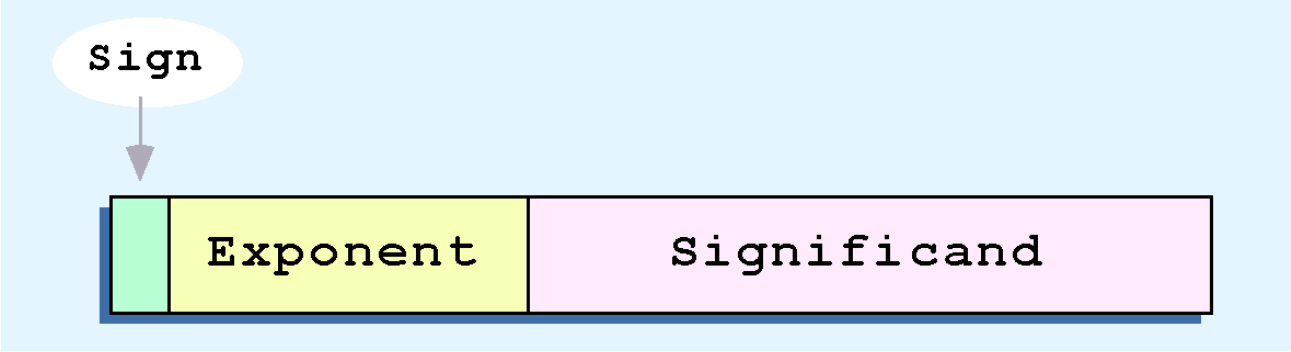
- The one bit *sign* field is the sign of the stored value.
- The size of the *exponent* field determines the **range** of values that can be represented
- The size of the *significand* determines the **precision** of the representation



Simplified Floating-Point Model

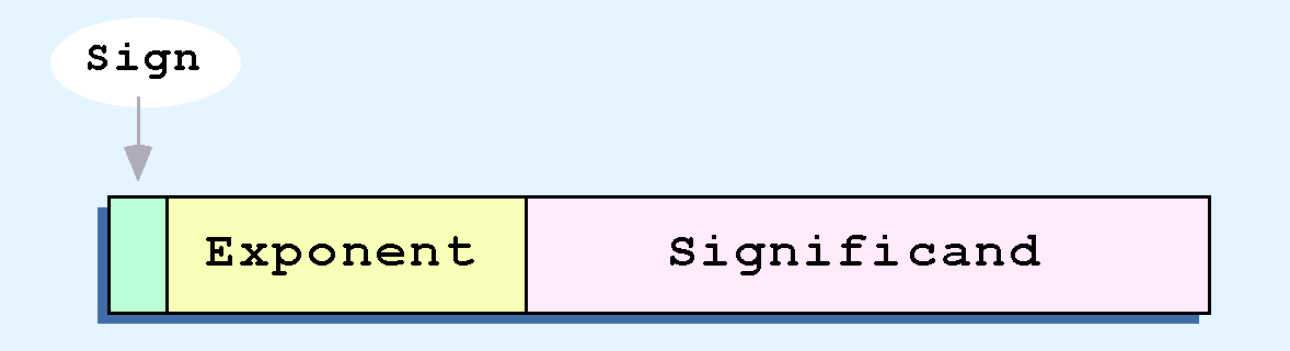
Floating-Point Representation

- We introduce a hypothetical "Simple Model" to explain the concepts with smaller numbers
  - *Later we'll discuss the real standard!*
- 14 bit long floating-point number:
  - The exponent field is 1 bit
  - The exponent field is 5 bits
  - The significand field is 8 bits



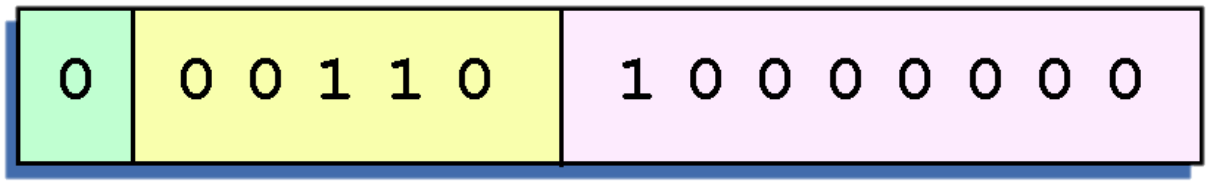
Floating-Point Representation

- The significand is always preceded by an implied binary point, i.e. `0.xxxxxxxxxx`
  - Thus it always contains a **fractional** binary value
- The exponent indicates the power of 2 by which the significand is multiplied



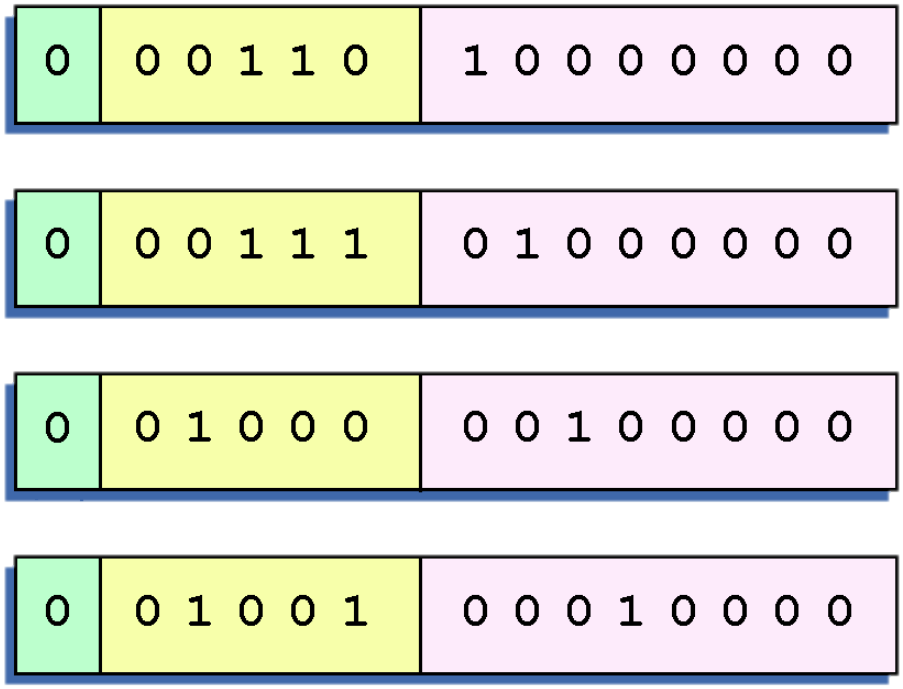
Floating-Point Representation

- Example: Express 32_{10} in the simplified 14-bit floating-point model
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$
  - In a moment, we'll explain why we prefer the second notation versus the first
- Using this information, we put 110 (= 6_{10}) in the exponent field and 1 (padded with 0's) in the significand:



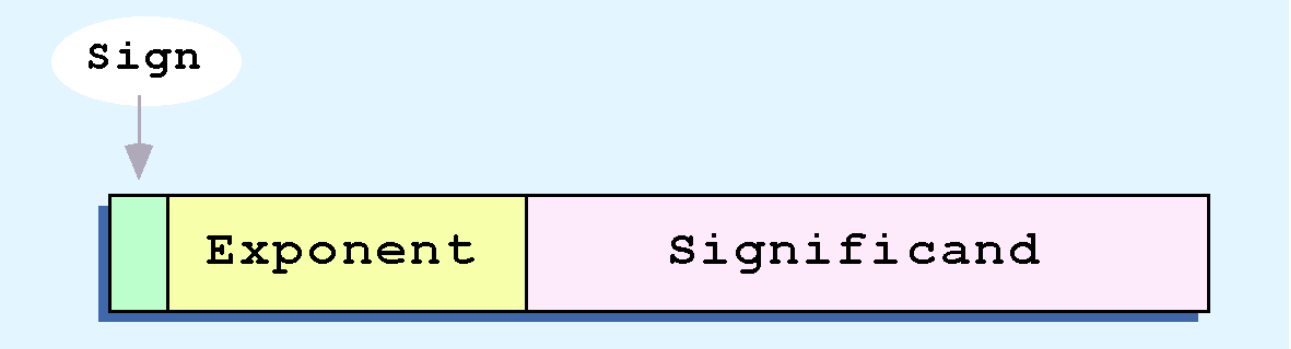
Floating-Point Representation

- **Problem 1:** We have many ways to represent the exact same number
  - Waste space
  - Cause confusion
  - Difficult to test for equality
- Figure: Multiple ways to represent 32 using our simplified model
  - By varying the exponent and shifting



Floating-Point Representation

- **Problem 2:** No way to express negative exponents
  - Can't store 0.5 ($=2^{-1}$)!
  - The exponent field is an **unsigned** value



All of these problems can be fixed with no changes to our basic model

Floating-Point Representation

- **New Rule #1**
  - The **first digit of the significand must be 1**
  - **In our simple model, all significands must have the form 0.1xxxxxxxx**
  - No ones to the left of the radix point
- This process is called **normalization**
  - Produces a unique pattern for each floating-point number
- Example: 4.5_{10}
  $= 100.1 \times 2^0$
  $= 1.001 \times 2^2$
  $= 0.1001 \times 2^3$.
  The last expression is correctly normalized.

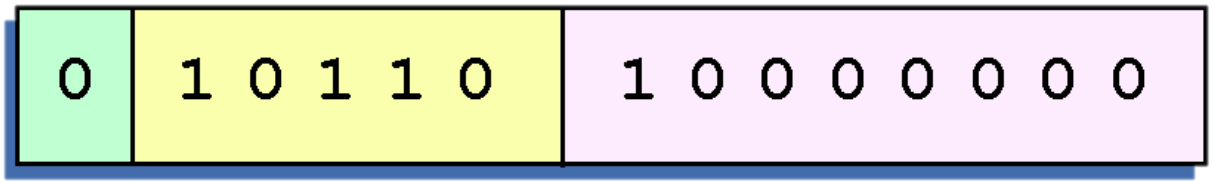
In our simple instructional model, we use no implied bits

Floating-Point Representation

- **New Rule #2:** To allow for negative exponents, we will use a *biased exponent*
- What is a *bias*?
  - A number that is approximately midway in the range of values expressible by the exponent
  - To determine the exponent to store, add the bias to your exponent
  - To decode a floating-point number, subtract the bias from the value in the exponent field
- Simplified model with 5-bit exponent
  - Use a bias of 16 (called an *excess-16* representation)
  - Exponent values less than 16 are negative, representing fractional numbers

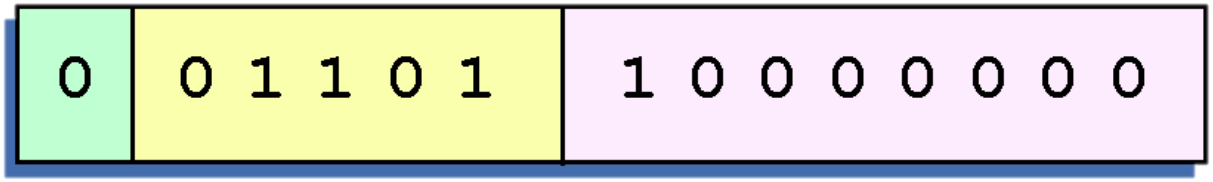
Floating-Point Representation

- Example: Express $\mathbf{32_{10}}$ in the revised 14-bit floating-point model
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$
  - Significant is now normalized (0.1xxxxxx)
- Use excess 16 biased exponent
  - Add 16 to 6 = 22_{10} (=10110_2)
- Final value saved to memory:



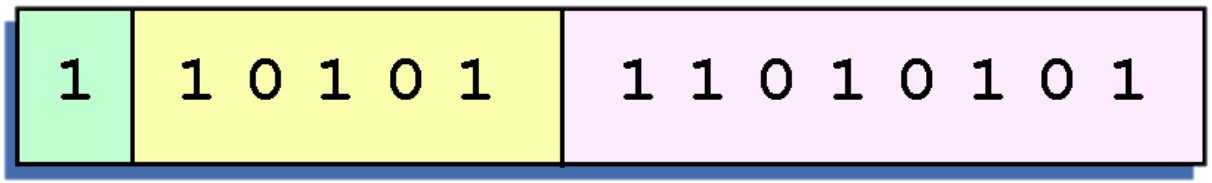
Floating-Point Representation

- Example: Express **0.0625_{10}** in the revised 14-bit floating-point model
- 0.0625 is 2^{-4}. In (binary) scientific notation, $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$ *(normalized notation)*
- Use excess 16 biased exponent
  - $16 + -3 = 13_{10}$ $(=01101_2)$
- Final value saved to memory:



Floating-Point Representation

- Example: Express **-26.625_{10}** in the revised 14-bit floating-point model
- $26.625_{10} = 11010.101_2 \times 2^0$
  Normalize = 0.11010101×2^5.
- Use excess 16 biased exponent:
  - $16 + 5 = 21_{10}$ ($=10101_2$)
- Also need a 1 in the sign bit (negative number)
- Final value saved to memory:



The **Real** Floating-Point Model

IEEE Floating-Point Representation

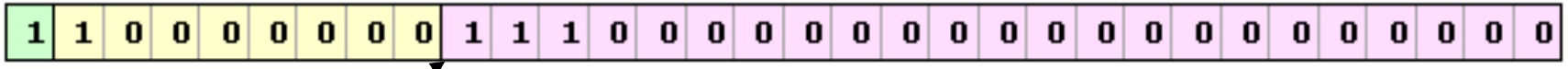
- The IEEE has established standards for floating-point numbers
- **IEEE-754 single precision** standard (32 bits long)
  - 8-bit exponent (with a **bias of 127**)
  - 23-bit significand
  - A “float” in C++
- **IEEE-754 double precision** standard (64 bits long)
  - 11-bit exponent (with a **bias of 1023**)
  - 52-bit significand
  - A “double” in C++

IEEE Floating-Point Representation

- **Key difference: Significand is normalized differently**
  - **Implied 1** to the **left** of the radix point, i.e. formatted as 1.xxx…
  - For example, $4.5 = .1001 \times 2^3$ in IEEE format is $4.5 = 1.001 \times 2^2$
  - The 1 is **implied**, which means it **is not saved in computer memory**
    - The stored significand would include only 001
    - Optimization – This saves a bit!

IEEE Floating-Point Representation

- Example: Express **-3.75** as a floating point number using IEEE *single* precision.
- Normalize according to IEEE rules:
  - $-3.75 = -11.11_2 = -1.111 \times 2^1$
  - The bias for *single precision* is 127, so add 127 + 1 = 128
    - This is the exponent saved to computer memory
  - The first 1 in the significand is implied, so we have:



(implied 1. not saved)

- To decode saved number with the implied 1 in the significand:
  - $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

IEEE Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
  - An exponent of 255 indicates a special value.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, “not a number,” often used to flag an error condition.
- Using the double precision standard:
  - An exponent of 2047 indicates a special value

IEEE Floating-Point Representation

- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero
  - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1
- Programmers should avoid testing a floating-point value for equality to zero
  - Negative zero does not equal positive zero

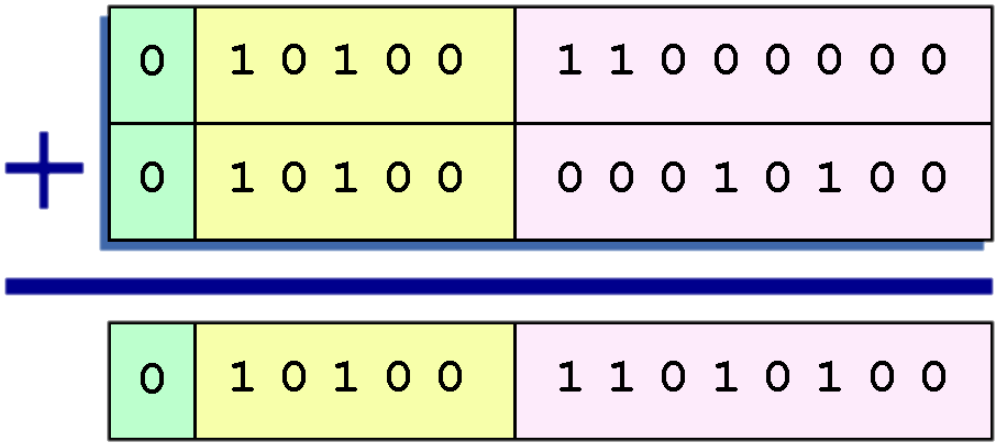
Floating-Point Representation

- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum
- If the exponent requires adjustment, we do so at the end of the calculation

Floating-Point Representation

- Example: Find 12_{10} + 1.25_{10} using the 14-bit simple floating-point model
- $12_{10} = 0.1100 \times 2^4$
  $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$

- Thus, the sum is 0.110101×2^4



Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
  - Multiply the two significands
  - Add their exponents
- If the exponent requires adjustment, do so at the end of the calculation

Floating-Point Errors

Floating-Point Errors

- No matter how many bits we use in a floating-point representation, our model is finite
- Problem: Real numbers can be infinite, so our model can only *approximate* a real value
- At some point, every model breaks down, introducing errors into the calculations
- By using a greater number of bits in the model, we can reduce these errors, but we can never totally eliminate them

Floating-Point Errors

- Goal: **Reduce error** (or understand how much error is present)
- Errors can compound through repetitive arithmetic operations.
- Example: The 14-bit model cannot exactly represent the decimal value 128.5
  - In binary, it is 9 bits wide: $10000000.1_2 = 128.5_{10}$
  - But we only have an 8-bit significand!

Floating-Point Errors

- How much error occurs when 128.5_{10} is represented with the 14-bit model?
  - True number: 128.5
  - Approximated number: 128
  - Error (percent difference)

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- If you wrote a loop that repetitively added 0.5 to 128.5 using 14-bit floating point, you would have an error of **nearly 2%** after only four iterations
  - *The error is less with "real" 32/64-bit floating point standards, but still exists*

Floating-Point Errors

- **Tip:** You can reduce floating-point errors by using operands that are similar in magnitude.
  - Instead of adding 128.5 + 0.5 + 0.5 + 0.5 + ….
  - Do this: 0.5 + 0.5 + 0.5 + …. + 128.5 (at the end!)
- In this example, the error was caused by loss of the low-order bit
- Loss of the high-order bit is more problematic

Floating-Point Errors

- Floating-point overflow and underflow can cause programs to crash
- Overflow occurs when there is no room to store the high-order bits resulting from a calculation
- Underflow occurs when a value is too small to store, possibly resulting in division by zero

Floating-Point Errors

- When discussing floating-point numbers, it is important to understand the terms *range*, *precision*, and *accuracy*
- The **range** of a numeric integer format is the difference between the largest and smallest values that can be expressed
- **Accuracy** refers to how closely a numeric representation approximates a true value
- The **precision** of a number indicates how much information we have about a value

Floating-Point Errors

- Most of the time, greater precision leads to better accuracy, but this is not always true
  - For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision

Floating-Point Errors

- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive
  - (a + b) + c = a + (b + c) or
  - a*(b + c) = ab + ac

May not be true!!

Floating-Point Errors

- To test a floating point value for equality to some other number, it is best to declare a “nearness to x” epsilon value
- Example: instead of checking to see if floating point x is equal to 2 as follows:
  - if (x = 2) then …
  - Do this:
    - if (abs(x - 2) < epsilon) then ...
    - Must define epsilon to be small, but not too small!

Character Codes

Character Codes

- Goal of signed/unsigned/floating-point systems
  - Store numbers for **computer** processing
- Goal of character codes
  - Store numbers/characters in the machine, but for **humans**
- Character codes have evolved alongside computers
  - Larger memories and storage devices permit richer character codes
  - The earliest computer coding systems used six bits
    - Fewest bits possible to represent digits (0-9), capital letters (A-Z), symbols

Character Codes - BCD

- Only 4 bits
- Sparse or **packed**
  - Sparse – Put one BCD digit in a byte / pad with 0's
  - Packed – Put two BCD digits in one byte

Digit	BCD (4 bits)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Unsigned Number	1111
Positive Number	1100
Negative Number	1101

Character Codes - EBCDIC

- Extended Binary-Coded Decimal Interchange Code (EBCDIC)
  - Introduced in 1964
  - Extension of BCD – 8 bits instead of 4
- EBCDIC was support amazing new features!
  - Upper **and** lowercase alphabetic characters!
  - Punctuation!
  - Control characters!
- EBCDIC and BCD are still in use by IBM mainframes today

Character Codes - ASCII

- American Standard Code for Information Interchange (ASCII)
  - Chosen by *non-IBM* companies
  - 7 bits – 8th bit could be used for **parity**
    - Even # of bits = 0, odd # of bits = 1
- ASCII was the dominant character code outside the IBM mainframe world
  - … until Unicode finally become widespread

Character Codes - Unicode

- What about **non-latin** characters?
- **Unicode**
  - 16-bits system allows for 65536 unique characters
  - All the Western (Latin, Cyrillic, Greek) characters, 4096 different symbols, 48,000+ Eastern (Chinese/ Japanese/Korean) characters, etc…
- Sufficient for **every** character of **every** language?
  - Not quite – but an optional extension allows for a million extra characters…

Character Codes - Unicode

- The Unicode codespace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code
- The highest provide for user-defined codes

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Error Detection / Correction

Error Detection and Correction

- **Why worry - aren't modern computers perfectly reliable?**
- 100% perfection is physically impossible for any data recording or transmission medium over its entire expected useful life
- 100% perfection gets harder to achieve as technology improves
  - Smaller bits on hard drive or memory – easier for cosmic rays to flip from 0<->1
  - Faster transmission of data on network – harder to tell the difference between a 0 and 1

Error Detection and Correction

- Check digits / parity bits – appended to the end of a long number – can provide some protection against data input errors
  - The last characters of UPC barcodes and ISBNs are check digits
- Longer data streams require more economical and sophisticated error detection mechanisms
- Cyclic redundancy checking (CRC) codes provide error detection for large blocks of data

Error Detection and Correction

- Data transmission errors are easy to fix once an error is detected
  - Just ask the sender to **re-transmit** the same data again
- In computer memory and data storage, however, this cannot be done
  - Too often the only copy of something important is in memory or on disk
- Thus, to provide data integrity over the long term, **error correcting codes** are required.

Error Detection and Correction

- **Hamming codes** and **Reed-Solomon codes** are two important error correcting codes
- Hamming codes can detect/correct a single bit error, and detect (but not correct) a two-bit error
  - Frequently used in RAM chips where errors are rare (but not rare enough!)
- Reed-Solomon codes are useful in correcting burst errors that occur when a series of adjacent bits are damaged
  - Scratch on a CD/DVD
  - Burst of static on a DSL phone line