CMSC 216
Introduction to Computer Systems
Data Representation

© UMD Department of Computer Science

Representing characters

- We need:
  - To be able to represent common characters
  - To have standards so computers can interoperate
- Common formats
  - **ASCII**
    - Uses 7 bits for characters (stored in 8 bits normally)
  - **EBCDIC**
    - An 8-bit code, used now only by some IBM mainframes
  - **UNICODE**
    - Provides a unique number for every character
    - A family of encodings - 8, 16, and 32 bits per character
    - Allows a greater variety of characters
    - Able to represent virtually any character in use today in any language, and some no longer in use
    - http://unicode.org/main.html
- Many 8-bit codes (such as ISO 8859-1, Linux default character set) have ASCII as their lower half

© UMD Department of Computer Science

ASCII

- Represents normal characters on US keyboards
  - **A-Z** (the characters numbered 65-90)
  - **a-z** (97-122)
  - **0-9** (48-57)
  - Space (32)
  - Control characters (0-31, 127)
    - First 26 (after 0) of the 33 ASCII control characters have names Ctrl-A - Ctrl-Z
  - Punctuation: **!@#$%^&*()_+-=[]{}|\;:"'<>?,./** (the remaining characters)
- The UNIX command "**man ascii**" displays the ASCII character set

© UMD Department of Computer Science

UNICODE

- **Different representations**
- UTF: Stands for **Unicode Transformation Format**
- UTF-32: a 32-bit representation of all characters
  - All characters are the same size
  - Uses lots of space (twice as much as UTF-16 for most things, four times as much as ASCII for many things)
- UTF-16: a 16-bit representation of characters
  - Some characters are stored in two-character forms
  - Popular since most things can be represented in 16 bits
- UTF-8: an 8-bit representation of characters
  - Provides backwards compatibility with ASCII
    - Low 7 bits are exactly ASCII
    - If the high bit is on it indicates part of UNICODE extensions
  - **Popular for web and other applications**

© UMD Department of Computer Science

Representing unsigned integers

- **All data** is stored in binary - all digits are 0 or 1
- In an unsigned number every bit position **i** represents the value $\mathbf{2^i}$, where **i** is 0 for the rightmost bit. The value of a number is the sum of the values of the bit positions containing a 1
- **Example bit position** values for an 8-bit number:

128	64	32	16	8	4	2	1

- The range of values that can be stored is 0 to $\mathbf{2^n - 1}$, where **n** is the number of bits used
  - For example, using 16 bits, the numbers 0 to 65,535 can be represented
- Addition of values can lead to overflow
- Overflow (cannot represent value):
  - When there is a carry out of the most significant bit position
    - How hardware can detect the problem

© UMD Department of Computer Science

Representing signed integers

- **We use two's complement representation for signed integers**
- In two's complement the leftmost bit represents the sign
  - Number is positive if leftmost bit is 0 and negative if it is 1
- **A positive integer value representation**
  - Same one use for an unsigned value (but leftmost bit is 0)
- **A negative integer value representation**
  - Flip all the bits of the corresponding positive value and add 1
    - Process referred to as **taking the two's complement**
    - This process also used to find the magnitude of a negative number (the corresponding positive number)
- Quick review of binary addition 0 + 0 → 0, 1 + 0 → 1, 1 + 1 → 0 (carry 1)
- **Example:** (assuming **4** bits used for signed integer representation):

```
+5 → 0101
     1010  /* Flipped Bits*/
      + 1  /* Adding One */
     1011
-5 (2's complement representation of -5) → 1011
```

- If we take the two's complement of -5 (1011), we get 0101

Representing signed integers

- If you add the previous two values +5 and -5:

+5 → 0101

-5 (2's complement) → 1011

0000 → Carry is dropped

- When you are representing an integer value in binary you need to know the number of bits you will use. The number of bits will determine the range of values you can represent
- **The range of an N-bit two's complement number is [-2^{N-1}, $2^{N-1}-1$]**
  - More negative numbers than positive numbers
- **Example:** For 4 bits the range will be -8 to 7
- Most negative number -2^{N-1}
- More negatives than positives as we don't have negative 0
- 0 is represented by all bit positions set to 0

© UMD Department of Computer Science

Representing signed integers

- **Overflow** (cannot represent expected value)
  - Adding two N-bit positive numbers or two N-bit negative numbers may cause overflow as the result may fall outside of the range for the N-bit two's complement number
  - Adding a negative and a positive number does not cause overflow
  - Overflow can be detected if the numbers added have the same sign and the result has the opposite sign
    - How hardware can detect overflow
- **Sign Extension**
  - When a two's complement number is extended to use more bits, the sign bit most be copied to the new bits position. For example, -5 two's complement (**1011)** extended to 8 bits will be **1111011**
  - **Example:** -1 (two's complement, 2 bits): **11**
  - How about -1 with 3 bits, 4 bits, 5 bits, etc. ? (111, 3-bits), (1111, 4-bits), …
  - **All 1's in a two's complement representation of a signed number is -1**
- **Reference:**
  - https://www.sciencedirect.com/topics/computer-science/twos-complement-number

© UMD Department of Computer Science

Representing signed integers

- Why we use 2’s complement?
  - Prevents the problem of two representations for zero associated with using storing numbers as unsigned with sign bit
  - The following is true: For all X, X + -X = 0
  - Makes hardware easy to develop as we can treat subtraction as addition
- **Example:** twos_complement.c

© UMD Department of Computer Science

How are floats/doubles represented?

- We have already seen the representation of signed/unsigned integers
- How about floats/doubles?
- A number can be represented as follows:

$$(-1)^{sign} \times mantissa \times radix^{exponent}$$, where r is the radix

- **Example:** The number $6132.789_{10} = 1 \times 6.132789 \times 10^{3}$ (radix is 10)
- **Example:** The number $0.05_{10} = 1 \times 5.0 \times 10^{-2}$ (radix is 10)
- **Example:** The number $-1001.1110_{2} = -1 \times 1.001110 \times 2^{3}$ (radix is 2)
- This is much like scientific notation, with the addition of the sign as a factor, and the ability to use a base other than 10

© UMD Department of Computer Science

Floating point representation, cont.

- Examples of floating point numbers

  $10.5_{10} = 1010.1_2 = .10101 \times 2^4$

  $7.4375_{10} = 111.0111_2 = .1110111 \times 2^3$ // Notice digits after period

- Decimal/binary points:

10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}
				•				
2^3	2^2	2^1	2^0		2^{-1} (1/2)	2^{-2} (1/4)	2^{-3} (1/8)	2^{-4} (1/16)

© UMD Department of Computer Science

The IEEE 754 floating point standard

- The IEEE 754 floating point standard has different sizes for values:
  - 32 bit floating point **(C float):**
    - 1 sign bit, 8 bits exponent, 23 bits mantissa
    - Range of representable values is approximately 2^{-126} .. 2^{128}, which is approximately 1.2×10^{-38} .. 3.4×10^{38}
  - 64 bit floating point **(C double):**
    - 1 sign bit, 11 bits exponent, 52 bits mantissa
    - Precision most commonly used for real applications
    - Range of representable values is approximately 2^{-1022} .. 2^{1024}, which is approximately 2.2×10^{-308} .. 1.8×10^{308}
  - 128 bit floating point **(quad):**
    - 1 sign bit, 15 bits exponent, 112 bits of mantissa
    - Not commonly used

© UMD Department of Computer Science

IEEE 754 floating-point numbers

- The leading 1 of the mantissa isn't stored:
  - The binary point (like a decimal point) is moved just to the right of the leftmost nonzero digit
  - But in binary, the leftmost nonzero digit must be a 1, so there's no need to actually store it, giving one more bit of precision in the mantissa for free
- The exponent:
  - Uses a bias, rather than two's complement, for storing negative as well as positive exponents. The bias is added to the exponent's value.
  - The bias is 127 for single-precision IEEE numbers (C **float'**s), and 1023 for double-precision numbers (C **double'**s)
- **The use of a bias allows the representation of the number zero to be all zeros;** in fact, **an exponent of all 1s or all 0s represents a special number**
  - 0, infinities, NaN, denormalized numbers

© UMD Department of Computer Science

Example IEEE floating-point number

- Here's how the example number -25.625 is represented in IEEE floating point (single precision (C float type)):
  - The sign bit (one bit) is 1, since the number is negative; we compute the absolute value of the number below
  - To compute the mantissa (23 bits):
    - Write the number in binary, with a binary point:

      25_{10} is 11001_2

      $.625_{10}$ is 1/2 + 1/8, which is $.101_2$

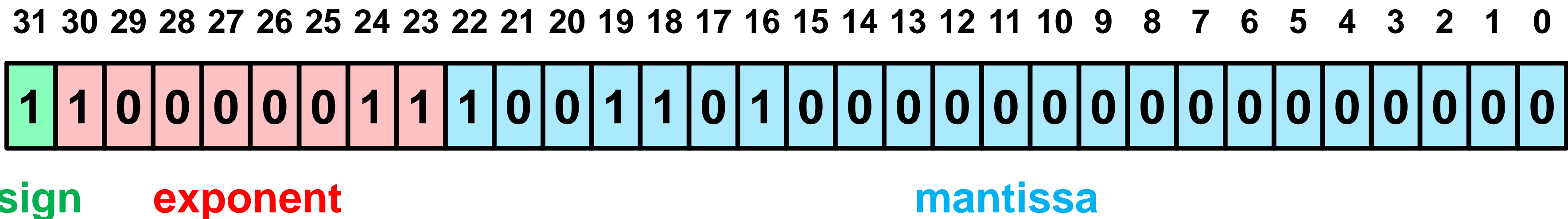
      so 25.625_{10} is 11001.101_2
    - Move the binary point right after the first nonzero digit, giving 1.1001101 (moved 4 places to the left)
    - Drop the leading 1 (and the binary point), giving **1001101**
    - Add zeros to the right to get 23 bits (here 16 zeros are needed)
    - So the mantissa is **1001101**0000000000000000

© UMD Department of Computer Science

Example IEEE floating-point number

- Recall the example number is -25.625
  - To determine the exponent (8 bits):
    - In the previous step, we moved the binary point 4 places to the left to place it to the right of the first nonzero digit, so the **exponent value is 4**
    - **To bias the exponent**, we add 127; 127 + 4 = 131, so the value of the exponent field is 131
    - 131 in binary is **10000011**
- Putting it all together, the number is represented as

  $(-1)^1 \times 1.1001101 \times 2^4 = -1.6015625 \times 16 = -25.625$
- And the number is stored in memory as



© UMD Department of Computer Science

Imprecision with real numbers

- The real numbers are dense (unlike the integers), but **anything in computer memory has to be stored in a finite bit representation**; this causes imprecision
- First consider an analogy with decimal numbers:
  - There are some numbers that can't be represented exactly in a finite number of digits - they require an infinite number of repeating digits
  - Example: 1/3 = .3333333333333...
  - Suppose we have only a fixed number of decimal digits in which to express 1/3, say for example 8 digits. The closest we can get is .33333333. But notice this is .0000000033333... away from the actual number 1/3
  - The next representable number (if we only have 8 digits) is .33333334, and any number between these two can only be approximated as one or the other of these two values- there are no values between them

© UMD Department of Computer Science

Imprecision with real numbers

- In binary there are also real numbers (not necessarily the same ones as in decimal) that can't be represented in a finite number of (binary) digits
- **Example:** $(1/3)_{10}$ = .0101010101010101010101...
- **Example:** $(1/5)_{10}$ = .00110011001100110011...
- If we have only four binary digits, the closest we can come to representing $(1/5)_{10}$ is .0011 (1/8 + 1/16 = .1875)
- If we have eight binary digits, we can come closer to representing $(1/5)_{10}$: .00110011 (1/8 + 1/16 + 1/128 + 1/256 = .19921875). The more digits we have, the closer we can come to representing it
- **But we'll never get exactly to 0.2_{10}, if we only have a fixed number of binary digits in which to represent the number**
- **Example:** imprecision.c

© UMD Department of Computer Science

Imprecision with real numbers

- The IEEE representation of 1/5, with a 23-digit mantissa, is 00111110010011001100110011001101, which works out to 0.20000000298023223876953125
  - See imprecision.c, entry float_value (26)
- The next smaller bit pattern (only one bit different) is 00111110010011001100110011001100, which works out to 0.19999998807907104492187500
- **There is no (single-precision) IEEE 754 float between these two values** because, with a fixed 23 digits of mantissa, there is no bit pattern between them
- If you try to compute or store values between these, such as 0.19999998825, 0.19999998850, 0.19999998875, etc., they'll all be represented as 00111110010011001100110011001100, which is 0.19999998807907104492187500
- We give up precision for magnitude

Another example (a large number)

- The IEEE float 375207297024.0 is represented as
  01010010101011101011100000110010
- The next bit pattern is 01010010101011101011100000110011
  which is the float 375207329792.0
- These two numbers are 32,768 apart, yet there is no IEEE 754 float value between them

© UMD Department of Computer Science

A 32-bit pattern can be …

- On a 32-bit machine, consider the bit pattern 11001101010101110101110000011001:
  - As an **unsigned integer**, this bit pattern represents the value **3445054489**
  - As a **two's complement signed integer**, this bit pattern represents the value **-849912807**
  - And as a **single-precision IEEE float**, this bit pattern represents the value **-225821072.0**
- **A pattern of bits can represent lots of different things** - we need to know what kind of thing they're supposed to represent to make sense of them
- **Example:** representation.c
- The following are not equivalent:

  *(float *)int_variable is **not** equivalent to (float)int_variable
- When you cast you perform a process that finds the value in a different representation

© UMD Department of Computer Science

Link

- Floating point math - http://0.30000000000000004.com/