Characters, Strings, and Floats

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- We need to represent textual characters in addition to numbers.
- Floating-point numbers provide a way to separate the magnitude of a number from the number of bits of significance it has.

B. Outcomes

At the end of today, you should:

- Know how textual characters are represented.
- Know why we have floating-point numbers and how they are represented.

C. Characters and Strings

- A character is represented by a bit string.
- ASCII is one scheme for characters; it uses 8 bits per character.
- The ASCII representations of
  - The digits 0–9 are hex 30–39 (decimal 48–57).
  - The letters A–Z are hex 41–5A (decimal 65–90)
  - The letters a–z are hex 61–7A (decimal 97–122)
  - The space character is hex 20 (decimal 32)
- ASCII C strings are null-terminated: They include an extra 8 bit of zeros.
  - **Example 1**: `"A9z"` is represented as the 4-character sequence `'A'`, `'9'`, `'z'`, `'\0'`. (The character `'\0'` represents 8 zero bits.) As a sequence of hex digits, this is `41 39 7A 00`. As a sequence of natural numbers ≥ 0 and < 256, we have 65 57 122 0.
- Unicode is a newer scheme that extends ASCII to use 16 bits per character.
  - Includes symbols from other languages and from math, logic, and other fields.

 © James Sasaki, 2015

- Strings are sequences of characters (null-terminated or length-specified).

D. Non-Whole Numbers

- In decimal, non-whole numbers are represented in various ways
  - **Example 2**: 3.75 and 3 ¾ and $3\ ^{75}/_{100}$ all represent the same number
- When using a decimal point, positions to the right of the decimal point indicate increasingly negative powers of 10: 10^{-1}, 10^{-2}, ….
  - **Example 3**: $3.75 = 3 \cdot 10^0 + 7 \cdot 10^{-1} + 5 \cdot 10^{-2}$
- Dividing by 10^n shifts the decimal point *n* digits to the left.
  - **Example 4**: 0.75 = 75 / 100, so $3.75 = 3\ ^{75}/_{100} = 3\ ¾$
- In binary, the positions to the right of the binary point indicate negative powers of 2.
  - **Example 5**: $1.011_2 = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$
    $= 1 + {}^1/_4 + {}^1/_8 = 1\ {}^3/_8 = 1.375_{10}$
- Dividing by 2^n shifts the binary point *n* bits left; multiplying by 2^n shifts right.
  - **Example 6**: $1.011_2 = (1011/1000)_2 = ({}^{11}/_8)_{10} = 1{}^3/_8$
- To convert a *decimal*.*fraction* to binary, you can treat the parts to the left and the right of the binary point separately. We already know how to deal with the part to the left. For the part to the right of the binary point, repeatedly subtract negative powers of 2.
  - **Example 7**: $1.375 = 1 + 0.375 = 1 + 0 \cdot 0.5 + 0.375$
    $= 1 + 0 \cdot 0.5 + 1 \cdot .25 + 0.125$
    $= 1 + 0 \cdot 0.5 + 1 \cdot .25 + 1 \cdot 0.125$
    $= 1.011_2$
- A **rational number** has the form numerator/denominator. It's the ratio of two integers. There are a couple of techniques for converting a decimal rational number to binary.
  - **Technique 1**: Convert the rational to decimal.fraction form first.
    - **Example 8**: $1{}^3/_8 = 1.375 = \ldots = 1.011_2$

 © James Sasaki, 2015

- **Technique 2**: If the denominator is a power of 2, say 2^n, then you can convert the numerator to binary and shift the binary point *n* bits left.
  - **Example 9**: $1^3/_8 = (8+3)/8 = {}^{11}/_8 = (1011_2)/2^3 = 1.011_2$

E. Scientific Notation

- One standard way to represent real numbers is scientific notation, as in 6.02×10^{23} or $\pm 1.1001_2 \times 2^{-4}$. In **normalized scientific notation**, we keep one (non-zero) digit to the left of the radix point (so 6.02×10^{23}, not 60.2×10^{22} or 0.602×10^{24}).
  - The overall sign, of course, says whether the number is positive or negative (its direction from the origin).
  - The exponent of a scientific notation number specifies its basic **magnitude** (its distance from the origin). Exponents > 1 are for values greater than the radix (or less than the negative of the radix). Exponents < 1 are for values between 0 and radix (or –radix and 0).
  - The **significand** or **mantissa** of a scientific notation number is the digit dot fraction part. To specify a number precisely, we need some number of **significant digits** (the significand has a required length).
- **Example 10**: For $1.011_2 \times 2^{-48}$, we need 4 significant bits. For $1.0110_2 \times 2^{-48}$ or $1.0111_2 \times 2^{-48}$, we need 5 significant bits.

F. Floating-Point Numbers

- In hardware, we use **floating-point numbers** to represent real numbers using scientific notation where the significand and exponent have fixed maximum sizes.
- So a floating point number might only be an approximation to an actual real number.
- Nowadays, people generally use an IEEE representation for floating-point numbers.
  - IEEE = Institute of Electrical & Electronics Engineers.
- For the IEEE representation, we break up a floating-point number into a sign bit *S*, an exponent field *E*, and a fraction field *F*. If the sign *S* is 1, the number is negative; if it is 0, the number is non-negative.
- The IEEE representation differs from normalized scientific notation in two ways:

 © James Sasaki, 2015

  - The IEEE fraction part omits the leading 1 dot. It's understood to be there but it's not written.
  - The IEEE exponent is written using a binary offset.
- Floating-point numbers come in various sizes. For a 32-bit IEEE floating-point number, the exponent is 8 bits long and the fraction is 23 bits long.
  - To get the scientific notation significand, we prepend a 1 dot to the 23-bit IEEE fraction, so we have 24 significant bits of precision.
  - If we read the IEEE exponent field E as an unsigned integer (where $1 \le E < 255$), then the scientific notation exponent is $E - 127$. So we can represent exponents ≥ -126 and ≤ 127. (The cases $E = 0$ and 255 are treated specially; see below.)
  - So the 32-bit string $S\ E\ F$ represents $(-1)^S \cdot 1.F \times 2^{E-127}$ where $1.F$ means "1." prepended to F.
- **Example 12**: Let N be the 32 bits 1100 0101 1011 0100 0000 0000 0000 0000. The sign field S is 1, the exponent field E is $1000\ 1011 = 139_{10}$, and the 23-bit fraction F is $011010^{(18)}$ where the notation $0^{(18)}$ means 18 zero bits. The scientific notation exponent is $E - 127 = 139 - 127 = 12$. The scientific notation fraction is $1.F$, which is $1.011010^{(18)}$. So N represents -1.01101×2^{12} (or $-1.011010^{(18)} \times 2^{12}$ if you want to write the trailing zeros).
- **Example 13**: In the other direction, if we are asked for the IEEE representation of -1.01101×2^{12}, we use 1 for the negative sign, drop the 1. and use $011010^{(18)}$ for the fraction, and add 127 to the exponent 12 to get 139 which is $1000\ 1011_2$. This gives $1\ 1000\ 1011\ 011010^{(18)}$ as the IEEE representation.
- The IEEE representation has some special cases.
  - If $E = 0$ and $F = 0^{(23)}$ we have $+0$ or -0 depending on the overall sign.
  - If $E = 0$ and $F \ne 0^{(23)}$ we have $N = (-1)^S \cdot 0.F \times 2^{-126}$. These are unnormalized scientific notation numbers, since they begin with 0; they're used for numbers that are extremely close to zero.
  - $E = 255$ is used for $+\infty$, $-\infty$, and *NaN* (Not a Number).

 © James Sasaki, 2015

G. Representation Problems With Floating-Point Numbers

- Our representation of floating-point numbers may fail in three ways. Two of them are magnitude problems; the third involves the number of significant bits.
- **Overflow**: The exponent is too large (i.e, too positive).
  - For 32-bit floating-point numbers, we can have $E = 254$ (so the actual exponent is 127). Writing $1^{(23)}$ to mean twenty-three 1 bits, the furthest we can get from zero is $\pm 1.1^{(23)} \times 2^{127}$.
- **Underflow**: The exponent is too small (i.e., too negative).
  - For 32-bit floating-point numbers, we can have $E = 1$ (so the scientific notation exponent is –126), so $\pm\ 1.0^{(23)} \times 2^{-126}$ is as close to zero as we can get using the standard case for IEEE floating-point numbers.
  - Unnormalized numbers (those that begin "0.") do let us get closer to zero. To get these numbers, we use $E = 0$ (and a nonzero significand). We get a scientific notation exponent of -126, so the smallest number we can represent has the IEEE representation S 0000 0000 $0^{(22)}1$. In scientific notation, we get $\pm 0.0^{(22)}1 \times 2^{-126}$, which equals $\pm\ 2^{-149}$.
- **Loss of Precision:** Since we only have so many bits for the fraction, we can only write numerals with a limited number of significant bits.
  - Since we are writing values in scientific notation, magnitude and significance are different.
  - **Example 14:** 1.01, 10.1, 101., 1010., 10100., … and .101, .0101, .00101, … all require 3 significant bits because we can write them using the form 1.01×2^N.
- Limiting the number of significant bits creates a limit to how close two floating-point numbers can be.
  - **Example 15**: with 32-bit IEEE floating-point numbers, we can't represent any number strictly between $1.0^{(22)}0$ and $1.0^{(22)}1$ because we would need > 24 significant bits. (By the way, that these two numbers only differ by 2^{-23}, so they're very close.)

 © James Sasaki, 2015

- **Example 16**: 2^{23} and $2^{23}+1$ are adjacent (we can't represent any number between them). We have $2^{23} = 1.0^{(23)} \times 2^{23}$ and $2^{23} + 1 = 1.0^{(22)}1 \times 2^{23}$. Note these values differ by 1, so they're much further apart than the ones in the previous example.

H. Arithmetic Problems With Floating-Point Numbers

- Because we can only represent certain real numbers with floating-point numbers, arithmetic operations can produce inaccurate results.
- **Overflow**: Adding or multiplying two large numbers can yield a result with an exponent that's too large.
  - **Example 17**: $2^{127} \times 2$ should $= 2^{128}$, which isn't representable using 32-bit floating-point numbers.
- **Underflow**: Dividing two numbers can yield an exponent that's too small.
  - **Example 18**: $2^{-126}/2$ should equal 2^{-127}, which isn't representable using 32-bit floating-point numbers.
- **Loss of Significance**: When we do arithmetic with two floating-point numbers, the result might require too many significant bits to represent.

Examples of Loss of Significance

- For examples of how we can lose significant digits, let's look at 5-bit floating-point numbers of the form $b.bbbb \times 2^N$ (where each b stands for a bit)
- **Example 19**: If we add 1.0000 and 0.0001, the result is 1.0001 and no loss of significance has occurred. Note we really had 1.0000×2^0 and 0.0001×2^0, so the exponents matched.
- In general, we may want to add two numbers where the exponents don't match. To make the exponents match, let's raise the smaller exponent to match the larger one. To do this, we shift its bits rightward, introducing 0s on the left and dropping bits on the right. If we drop a 1 bit, we're losing a significant digit. (Shifting out bits that are 0 seems less of a problem.)
- **Example 20**: Take $1.0000 \times 2^1 + 1.0000 \times 2^{-2}$
  - First shift $1.0000 \times 2^{-2} = 0.1000 \times 2^{-1} = 0.0100 \times 2^0 = 0.0010 \times 2^1$

 © James Sasaki, 2015

  - Note we lost the two rightmost 0 bits.
  - Now add $1.0000 \times 2^1 + 0.0010 \times 2^1 = 1.0010 \times 2^1$
- **Example 21**: $1.0000 \times 2^1 + 1.0011 \times 2^{-2}$.
  - This time we shift 1.0011×2^{-2} to 0.0010×2^1, but we lose the two rightmost 1 bits to do it.
  - Then we add $1.0000 \times 2^1 + 0.0010 \times 2^1 = 1.0010 \times 2^1$
- Note in general, floating-point addition isn't associative because we may lose significance when adding numbers with dissimilar exponents.
- Similar truncation problems occur with other arithmetic operations. Because each operation can introduce an error, it's tricky to write algorithms that do calculations using floating-point numbers that keep the maximal number of significant digits.

 © James Sasaki, 2015

Characters, Strings, and Floats

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- We need to represent textual characters in addition to numbers.
- We use floating-point numbers to represent non-whole numbers (numbers not evenly divisible by 1).

B. Outcomes

After this activity, you should

- Be able to describe the difference between characters and their ASCII representations.
- Be able to represent floating-point numbers in binary and using the IEEE representation.

C. Problems

1. Let `'0'`, `'1'`, …, `'9'` be the ASCII representations of the digits 0 – 9. (In C, you can get these numbers using `(int) '0'`, etc.)

   (a) What are the values of `'0'`, `'1'`, …, `'9'`? Which (if any) of the following are true? (b) `'2'+'3' = '5'` (c) `'2'+3 = '5'` (d) `2+'3'= '5'` (e) `2+3 = '5'`.

2. Let N be an integer with $0 \leq N \leq 9$. What are the integer values of the following: (a) `'0'+`N (b) `'0'+'N'` (c) `'A'+`N (d) `'a'+`N (e) Which of (a) through (d) represent printable characters? What are those characters?

3. Let X represent one of the capital letters A – Z. What are the integer values of: (a) X`-'A'` (b) X`-'A'+'a'` ? (c) Which of (a) and (b) represent printable characters? What are those characters?

4. What decimal number does 10.011_2 represent? (There are multiple ways to write the answer.)

 © James Sasaki, 2016

5. Let $X = 6.4375_{10}$.
   (a) What is the binary representation of X?
   (b) What is its scientific notation representation? (I.e., 1.something × 2 raised to some power.)
   (c) What is its 32-bit IEEE representation?
6. Let $Y = 10.011_2$.
   (a) What is the scientific notation representation of Y?
   (b) What is the 32-bit IEEE representation of Y?
   (c) What is the hex representation of the result from (6b)? Hint: You need 8 hex digits to represent the 32 bits.
7. On average, the Moon is $D = 238{,}000$ miles from the Earth. (Note: $2^{17} < D < 2^{18}$.)
   (a) If we represent D using the IEEE 32-bit format, what is the largest error you could expect when trying to represent distances in miles?
   (b) If you were launching a lunar probe, would you want to use 32-bit floating-point numbers in your navigation system?
8. Let's assume that we're doing addition using 5 significant bits and that if we need to lose precision, we truncate the offending bits.
   (a) If we add 1.1111 + 0.1110 using 5 significant digits, does truncation (of nonzero bits) occur before the addition? After the addition?
   (b) Repeat, for 1.1001 + .11101 (by .11101 I mean 1.1101×2^{-1}).*
   (c) Repeat, for 1.1101 + .01100.
9. In general, is floating-point addition associative?

* If we have n significant bits and $\leq n$ bits to the right of the binary point, we can put the binary point inside the bitstring.

 © James Sasaki, 2016