

# ***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)
- 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.
  - Strings are sequences of characters (null-terminated or length-specified).
- ASCII C strings are null-terminated: They include an extra 8 bit of zeros.
  - E.g., "A9z" would be hex 41 39 7A 00.
  - The character representation of 00<sub>16</sub> (equals 0000 0000<sub>2</sub>) is '\0'.

### ***D. Non-Whole and Floating-Point Numbers***

- In decimal, non-whole numbers are represented using a decimal point and a fractional part that sums up negative powers of 10.
- In binary, non-whole numbers use a binary point and negative powers of 2.
  - E.g.,  $1.01101_2 = 1 + 1/4 + 1/8 + 1/32 = 1 \text{ }^{13}/_{32} = 1.40625_{10}$
- True real numbers can take an infinite number of digits.
  - We approximate them with a finite number of digits.
  - Tradeoff representing numbers with a large magnitude vs numbers with a lot of precision. **Magnitude:** Distance from zero. **Precision:** Number of significant digits. The **range** of a set of numbers is the distance between the most positive and most negative numbers.
- Floating-point numbers use separate bits for magnitude and precision
  - Based on scientific notation. E.g.  $1.011_2 \times 2^{-56}$  or  $-1.101_2 \times 2^{12}$ .
  - Note 1: The bit to the left of the binary point will always be 1.
  - Note 2: The overall sign of the number and the sign of the exponent can be different.

### ***E. IEEE Floating-Point Number Standard***

- IEEE = Institute of Electrical & Electronics Engineers.
- Break up 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.
- Floating-point numbers come in various sizes. For a 32-bit IEEE floating-point number, the exponent is 8 bits and the fraction is 23 bits.
- The IEEE representation differs from scientific notation in two ways: (1) The fraction part omits the leading 1. — it's understood to be there but it's not written; (2) The exponent field is 127 plus the scientific notation exponent.
- **Example:** 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, and the 23-bit fraction  $F$  is 01101 00000 00000 00000 000. Since  $E = 1000 1011 = 139_{10}$  the actual

exponent is  $139 - 127 = 12$ . To get the actual fraction, we prepend "1." to  $F$  and get  $1.011010...0$ . So  $N$  represents  $-1.01101 \times 2^{12}$ .

- **Example:** For the other direction, if we were asked for the IEEE representation of  $-1.01101 \times 2^{12}$ , we'd use 1 for the negative sign, drop the 1. and use 01101...0 for the fraction, and add 127 to the exponent 12. Since  $12 + 127 = 139 = 1000\ 1011$  for the exponent; this gives 1 1000 1011 01101 followed by eighteen 0's.
- In general, the case where  $1 \leq E < 255$  is the standard case above.
  - In this case,  $S\ E\ F$  represents  $(-1)^S \times 1.F \times 2^X$  where  $X = E - 127$ .
- There are some other cases (these won't be on the tests):
  - The case where  $E = 0$  is used for floating-point zero and for extremely small numbers (very close to zero).
    - If  $E = 0$  and  $Fraction = 0$  we have  $+0$  or  $-0$ .
    - If  $E = 0$  and  $Fraction \neq 0$  we have  $N = (-1)^S \times 0.Fraction_2 \times 2^{-126}$   
(\*\*\*)
  - The case where  $E = 255$  used for  $+\infty$ ,  $-\infty$ , and  $NaN$  (Not a Number).  
(\*\*\*)

## Other Data

### *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. Questions**

1. Let '0', '1', ..., '9' be the ASCII representations of the digits 0 – 9. (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  $\geq 0$  and  $\leq 9$ . What are the integer values of the following: (a) '0'+ $N$  (b) '0'+'N' (c) 'A'+ $N$  (d) 'a'+ $N$  (e) Which of (a) – (d) represent printable characters and 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 and what are those characters?
4. What decimal number does  $10.011_2$  represent?
5. (a) What is the binary representation of  $6.4375_{10}$ ? (b) What is its scientific notation representation? (I.e., 1.something  $\times 2$  raised to some power.)
6. What are the scientific notation and IEEE 32-bit representations for  $10.011_2$  ?

7. (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  $1.1001 + .11101$ . (c) Repeat, for  $1.1101 + .01101$ .
8. In general, is floating-point addition associative?

**Solution**

- 1a. '0' = 48, '1' = 49 ..., '9' = 57.
- 1b. '2' + '3' = 50 + 51 = 101 is  $\neq$  53 = '5'
- 1c. '2' + 3 = 50 + 3 = 53 is = '5'
- 1d. 2 + '3' = 2 + 53 = 55 is = '5'
- 1e. 2 + 3 = 5 is  $\neq$  53 = '5'.
2. Given  $N$  is an integer,  $0 \leq N \leq 9$ , we have
  - 2a. '0' +  $N$  = 48 +  $N$  = the ASCII representation of  $N$
  - 2b. '0' + 'N' = 48 + 78 = 126 (which happens not to = '~')<sup>1</sup>.
  - 2c. 'A' +  $N$  is the representation of the  $N$  + 1st capital letter of the alphabet,  
(If  $N = 0$  then 'A' +  $N$  = 'A', if  $N = 2$ , then 'A' +  $N$  = 'B', etc.)
  - 2d. 'a' +  $N$  is the representation of the  $N$  + 1st lowercase letter of the alphabet.  
(If  $N = 0$  then 'a' +  $N$  = 'a', if  $N = 2$ , then 'a' +  $N$  = 'b', etc.)
  - 2e. (See answers above.)
- 3a. If  $X$  is the  $n$ th letter of the uppercase alphabet ( $1 \leq n \leq 26$ ) then  $X = \text{'A'} + n - 1$ , so  $X - \text{'A'} = n - 1$ . (If  $X = \text{'A'}$ , then  $X - \text{'A'} = 0$ ; if  $X = \text{'B'}$ , then  $X - \text{'A'} = 1$ ; ..., if  $X = \text{'Z'}$ , then  $X - \text{'A'} = 25$ .)
- 3b.  $(X - \text{'A'}) + \text{'a'} = n - 1 + \text{'a'}$ . This sum is the lowercase version of the uppercase  $X$ .
4.  $10.011_2 = 2 + .011_2 = 2 + (11_2/100_2) = 2^3/8$ . Also,  $2 + .011_2 = 2 + .25 + .125 = 2.375$  (which =  $2^3/8$ , of course).
- 5a.  $6.4375_{10} = 6 + 0.25 + .125 + .0625 = 110.0111_2$ .
- 5b.  $110.0111_2 = 1.100111_2 \times 2^2$
6.  $10.011_2 = 1.0011_2 \times 2^1$ . For the IEEE representation, the sign  $S = 0$  (since the number is  $\geq 0$ ), the exponent  $E = 127 + 1 = 128 = 1000\ 0000_2$ , and the fraction  $F$  is 1.0011 less the leading "1." plus enough zeros to make 23 bits, so  $F = 0011\ 000000\ 00000\ 00000\ 000$ . Combining, we get the 32-bit

---

<sup>1</sup> You're not expected to know 'N' = 78 or '~' = 126 for the tests.

representation: 0 1000 0000 0011 000000 00000 00000 000. (Not asked for: This is hex 40180000.)

- 7a. We don't need to drop any bits to align the binary points of 1.1111 and 0.1110 (so no truncation), but we have to convert the result  $10.1101 \rightarrow 10.110$  because we only have 5 significant digits, and we do truncate the rightmost 1.

```

  1.1111
+0.1110
-----
 10.1101

```

- 7b. Aligning  $1.1001 + .11101$  causes truncation of 1 in the right-hand value; we get  $1.1000 + 0.1110$ ; addition yields 10.0110, which turns into 10.011 (no truncation of nonzeros)
- 7c. Aligning  $1.1101 + .01101$  turns it into  $1.1101 + .0110$  (truncates a 1); addition yields 10.00111; we truncate 11 to get 10.001.
8. No, in general floating-point addition (and arithmetic in general) isn't associative.