Data Representation

Math for Computer Science – 201-H01-HR

Unit 1

1 Decimal Numbers

When using numbers in every day applications, we use the *decimal number system* (also called the *base-10 number system*). The decimal number system is over 1,000 years old and uses ten symbols to encode numbers, namely the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Decimal numbers are made up of digits which consist of the above symbols in specific positions. Each digit has a *face value* which is the digit itself and a *place value* which is a power of 10.

Ex: State the face value and place value of each digit in the number 276.

For decimal numbers with a *decimal point*, the digits to the right of the decimal point have place values which are powers of 10 with negative exponents.

Ex: State the face value and place value of each digit in the number 132.85.

The *expanded form* of a decimal number is the sum of the face value of each digit multiplied by its place value.

Ex: Write 821 and 24.179 in their respective expanded forms.

2 Binary Numbers

The *binary number system* (also called the *base-2 number system*) uses only the symbols 0 and 1 to encode numbers. A single binary digit is called a *bit* and a group of 8 bits is called a *byte*.

Bits can be used to model ...

- whether a switch in an electric circuit is open or closed.
- whether a capacitor in a microprocessor is holding a charge or not.
- whether a bump appears at a certain place on a DVD or not.
- whether a given statement is true or false.

Binary number systems are used by all modern computers and telecommunication devices.

In the binary number system, the face value of a bit is either 0 or 1 and the place value of a bit is a power of 2.

Ex: State the face value and place value of each bit in the binary number 101.011. Write this number in expanded form.

To easily distinguish between decimal numbers and binary numbers, we add a subscript of 10 or 2, respectively, to the end of the number. Any number whose base is not given is assumed to be base-10.

Ex: Write the expanded form of 1011_{10} and 1011_2.

A binary number can be converted to a decimal number by computing the sum of the expanded form of the binary number.

Ex: Convert each of the following binary numbers to decimal numbers.

1. 1101_2

2. 10.101_2

3 Octal Numbers

The *octal number system* (also called the *base-8 number system*) uses the symbols 0, 1, 2, 3, 4, 5, 6, and 7 to encode a number. Here the face value of a digit is one of the above symbols and the place value of a digit is a power of 8.

Octal number systems were used in earlier days of computing but are rarely used today. One application, however, is the use of octal numbers to set permissions with the `chmod` command used in the C-programming language or in Unix.

Ex: State the face value and place value of each digit in the number 732_8.

Ex: Write the expanded form of 15.3_8 and convert it to a decimal number.

4 Hexadecimal Numbers

The *hexadecimal number system* (also called the *base-16 number system*) uses the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F to encode a number. Here the face value of a digit is one of the above symbols and the place value of a digit is a power of 16. The symbols A, B, C, D, E, and F have values 10, 11, 12, 13, 14, and 15 respectively.

Digits of hexadecimal numbers are called *nibbles* because each nibble describes exactly one of the numbers in a half-byte.

Hexadecimal numbers are used ...

- as memory addresses in RAM.
- to encode colours for computer graphics in HTML code.
- in machine language or assembly language programming.

Ex: State the face value and place value of each nibble of $D39_{16}$.

Ex: Write the numbers $BEAD_{16}$ and $4.A3_{16}$ in expanded form.

Ex: Convert the number $83C_{16}$ to decimal.

Practice: Write each of the following numbers in expanded form and convert them to decimal.

1. $1DF_{16}$

2. 63.5_8

3. 101.011_2

5 Converting a Number in a Different Base to Decimal

You may use the method previously shown this week (multiply each face value by its place value and add up the results)

Another method to convert an **integer** number in base-n to decimal:

1. Let the current result be the leftmost digit.
2. Stop if there are no more unprocessed digits.
3. Multiply the current result by n.
4. Add the next digit to the current result and go to step 2.

Ex: Use the above method to convert 137_8 to decimal.

6 Converting a Decimal Number to Another Base

Method: Convert a decimal number to base-n.

1. Find the largest power of n that is not larger than the number.
2. By dividing, determine how many times this power of n goes into the number. The result is the next digit in the base-n representation.
3. Repeat steps 1 and 2 with the remainder until the number is completely converted, inserting zeroes as placeholders if any powers of n are skipped.

Ex: Convert 217 to hexadecimal.

Ex: Convert 61.5 to binary.

Another Method: To convert an **integer** decimal number to base-n, repeatedly divide the number by n keeping track of the remainders as shown in the examples below.

Ex: Convert 340 to octal.

Ex: Convert 79 to binary.

Converting a Decimal Fraction to Base-n: Apply the previous method to the integer part of the number. Take the purely fractional part and repeatedly multiply by n, removing each resulting integer part as a digit in the base-n representation as shown in the examples below.

Ex: Convert 0.6 to binary.

Ex: Convert 29.3 to octal.

Practice: Convert each of the following decimal numbers to the requested number system.

1. Convert 827 to hexadecimal.

2. Convert 87.3 to binary.

7 Converting Between Binary and Octal

To convert an octal number to binary, simply write each octal digit in binary.

Ex: Convert 531_8 and 46.3_8 to binary.

To convert a binary number to octal, group together the bits of the binary number in groups of 3, starting immediately to the left or to the right of the octal point and padding with zeroes when necessary. Convert each group of 3 bits into its corresponding digit.

Ex: Convert 10101_2 and 10.1101_2 to octal.

8 Converting Between Binary and Hexadecimal

To convert a hexadecimal number to binary, we simply write each nibble in binary. To convert a binary number to hexadecimal, we proceed as when converting a binary number to octal, except that we use groups of 4 bits.

Ex: Convert BEE_{16} and $3.A2F_{16}$ to binary.

Ex: Convert 10101_2 and 10.1101_2 to hexadecimal.

To convert between octal and hexadecimal, simply write the number in binary and then convert the binary number to the appropriate base.

Ex: Convert ACE_{16} to octal.

Ex: Convert 3.72_8 to hexadecimal.

Practice: Convert each of the following numbers to the requested base.

1. Convert 652.4_8 to binary.

2. Convert 110010_2 to hexadecimal.

3. Convert $A3.5F_{16}$ to octal.

9 Adding Numbers in Various Bases

To add numbers in base-n we do it exactly as you first learned addition, but carrying over digits whenever an individual sum is greater than or equal to n.

Ex: Perform the addition $357_8 + 210_8$.

Ex: Perform the addition $A5F_{16} + 3C_{16}$.

Ex: Perform the addition $11.011_2 + 1.101_2$

10 Subtracting Numbers in Various Bases

Subtracting numbers in base-n (provided the result is nonnegative) is exactly as you first learned it with decimal numbers. The only difference occurs when we borrow 1 from the digit to the left in the top row, it adds n to the current digit in the top row.

Ex: Perform the subtraction $6D29_{16} - 4B17_{16}$.

Ex: Perform the subtraction $1010_2 - 101_2$.

Ex: Perform the subtraction $76.3_8 - 7.45_8$.

11 Multiplying Numbers in Various Bases

Multiplying numbers in base-n is exactly as you first learned it with decimal numbers except that you must convert all intermediary products to base-n.

Ex: Perform the multiplication $1011_2 \cdot 10.1_2$.

Ex: Perform the multiplication $42_8 \cdot 75_8$.

Ex: Perform the multiplication $C2_{16} \cdot D.7A_{16}$.

12 Dividing Numbers in Various Bases

Dividing numbers in base-n is exactly as you first learned it except that all intermediary multiplications and subtractions must be in base-n.

Ex: Perform the division $473_8/6_8$.

Ex: Perform the division $10110_2/11_2$.

Ex: Perform the division $8E2_{16}/2A_{16}$.

If we multiply or divide numbers in base-n by a power of n, it is very easy to compute the result. All that changes is the location of the point.

Ex: Perform the multiplication $1011_2 \cdot 100_2$.

Ex: Perform the multiplication $AC14_{16} \cdot 100000_{16}$.

Ex: Perform the division $631.52_8 / 10_8$.

Practice: Compute each of the following.

1. $1011.101_2 + 110.01_2$

2. $372_8 - 65_8$

3. $4A.26_{16} - D.17B_{16}$

4. $735.61_8 \cdot 10000_8$

5. $3C_{16} \cdot A9_{16}$

6. $101.1101_2 \cdot 1.01_2$

7. $771_8 / 12_8$

8. $1100_2 / 101_2$

13 Fixed-point Representations of Numbers

Storing real numbers in a computer requires us to consider several practical issues. For instance, if our computer is binary, how will we represent a binary point or a negative number?

For a *fixed-point representation* of a number, the (binary or decimal) point is understood to be in a fixed position. Different locations for the point are achieved by using different data types. Fixed-point representations written in base-n (usually base-2 or base-10) express a rational number whose denominator is a fixed power of n.

Example: If we are storing a real number using a byte with a fixed-point representation where the binary point is understood to be immediately to the left of the 3rd bit (from the right) then 10110101 in memory corresponds to the number 10110.101_2. Similarly, this number can be thought of as $10110101_2/1000_2$.

Example: If we are storing an amount of money using a 9-digit decimal fixed-point representation where the decimal point is understood to be immediately to the left of the 2nd digit then 004256781 corresponds to the number $42,567.81_{10}$. Similarly, this number can be thought of as $4256781_{10}/100_{10}$.

For the remaining sections involving fixed-point representations, we will focus on 1-byte binary representations where the binary point is understood to be at the rightmost end of the number (i.e. the numbers being stored are integers).

Nonnegative numbers can be stored as *unsigned numbers* in a computer. A given unsigned number stored in a byte is simply storing a number equal to the byte's binary representation.

Example: The unsigned number 10110110 is $10110110_2 = 182_{10}$.

Ex: Determine the unsigned number which can be used to store 87_{10} in a byte of memory.

14 Signed Numbers

Integers can be stored as *signed numbers* in a computer. There are several different ways that have been used to accomplish this.

- Sign-and-magnitude
- One's Complement
- Two's Complement

One early attempt, called *sign-and-magnitude*, was to use the 7 rightmost bits in the byte to store the magnitude of the number and to use the leftmost bit of the byte to indicate the sign of the stored number. A 1 in the leftmost bit indicated that the number was negative whereas a 0 would denote that the number was positive. Zero could be represented as 10000000 (-0) or 00000000 (+0).

Ex: Write the sign-and-magnitude representations of the decimal numbers 14 and -14.

Although, intuitively following from the hand-written notation used for integers, sign-and-magnitude representations require complicated rules for arithmetic and are not generally used.

15 One's Complement

The *one's complement* representation of a negative number is found by taking the magnitude of the number written in binary and replacing each 0 bit with a 1 and vice versa. The one's complement representation of a positive number is simply its binary representation. This scheme is only used to represent numbers between -127 and 127. Zero has two one's complement representations. Interestingly, the leading bit of every negative number is 1 and the leading bit of every positive number is 0.

Ex: Write the one's complement representation of 53 and -53.

Addition can be performed with one's complement representations. It is exactly like binary addition except that if there is carryover past the 8th bit then a 1 is added to the total.

Ex: Add -34 + 19 using one's complement representations.

Ex: Subtract 18 - 51 using one's complement representations.

One's complement is not used very often today.

Practice:

1. Find the one's complement of -22 and -103.

2. Add -22 + 71 using one's complement representations.

3. Subtract 65 - 103 using one's complement representations.

16 Two's Complement

To represent an integer between -128 and 127, inclusive, in *two's complement form* we first find the binary representation of the absolute value of the number and write it in a byte. If the number is negative, we change all the 0's to 1's (and vice versa) and add 1 to the result.

As a consequence, each integer between -128 and 127, inclusive, has a unique two's complement form. Also, every negative number has a 1 as its leftmost bit and every nonnegative number has 0 as its leftmost bit. Hence the leftmost bit is called the *sign bit*.

Ex: Find the two's complement form of 83 and -57.

To add two integers written in two's complement form, we simply add them as we would two binary numbers. There is no need to add any extra rules. If the sum results in a carryover to a 9th bit, then this last bit is simply dropped.

Ex: Perform the addition $-57 + 83$ using two's complement form.

Since we can find the two's complement form of negative numbers and we can add, we can also subtract. In this way, we can subtract two integers **without "borrowing"** from bits to the left. Due to this simplicity, two's complement is the most common way computers use to store integers.

Ex: Perform the subtraction $61 - 99$.

Practice: Compute each of the following additions or subtractions using two's complement form.

1. $18 + 44$

2. $23 - 17$

3. $47 - 101$

17 Significant Figures

We will now turn our attention to how computers store real numbers.

When we write down a real number, we are giving information about the *significant figures* of the number. To begin with, the *most significant figure* is the leftmost nonzero symbol. The significant figures can be identified as follows.

- All non-zero symbols are significant.
- A zero anywhere between two significant figures is significant.
- All zeroes to the right of the radix point and the rightmost nonzero symbol are significant.
- Any zero to the left of the most significant figure is not significant.
- If the number is an integer, the zeroes in a block of zeroes immediately to the left of the radix point may or may not be significant.

Ex: Give the significant figures for each of the following numbers.

1. 0073_{10}
2. $0.02AF_{16}$
3. 10.000_2
4. 23000_{10}

Due to the ambiguity of this last example, we use the *floating point representation* for real numbers to clearly indicate the significant figures.

18 Floating Point Representation

A *floating point representation* of a real number is a way of representing the number as a product of a real number (called the *significand* or mantissa) with a power (containing a base and an exponent). The significand is chosen so that it accurately displays the significant figures of the original real number.

Ex: Write two different floating point representations for each of the following real numbers with the requested base.

1. 143.26_{10}

2. 00110101_2

3. $9.A00_{16}$

4. 43000_8 where this number has 4 significant figures.

Since every real number has many different floating point representations, *normalized notation* is used. The normalized notation of a real number is unique floating point representation where the most significant figure is immediately to the left of the radix point.

Ex: Write each of the following real numbers using normalized notation with the requested base.

1. $4D.73_{16}$

2. 101100_2

3. 0.000728_{10}

4. 30.000_8

19 IEEE Standards

The IEEE standard for floating point arithmetic (IEEE 784) is the most popular standard for storing real numbers in computer memory. Of course, the standard requires us to convert any real number to binary first. The two most common basic forms of this standard are *single precision* and *double precision.*

Single Precision form:

- 32 bits are used.
- Start with your number written in its binary representation and using normalized notation.
- The leftmost bit denotes the sign of the number (1 for negative and 0 for nonnegative).
- The next 8 bits store the exponent of the base 2 from normalized notation **plus** 127_{10}.
- The remaining 23 bits store the bits that come after the binary point of the significand. For normalized notation of binary representations, the only significant bit to the left of the binary point is a single 1 and so this bit is not stored in memory.

Ex: Write each of the following numbers in single precision form.

1. $-1.0101_2 \times 2^{111_2}$

2. $1.1011_2 \times 2^{-78}$

Practice: Convert each of the following decimal numbers to their binary representation using normalized notation. Then write them in single precision form.

1. 1043

2. 0.75

Double Precision form:

- 64 bits are used.
- Start with your number written in its binary representation and using normalized notation.
- The leftmost bit denotes the sign of the number (1 for negative and 0 for nonnegative).
- The next 11 bits store the exponent of the base 2 from normalized notation **plus** 1023_{10}.
- The remaining 52 bits store the bits that come after the binary point of the significand. For normalized notation of binary representations, the only significant bit to the left of the binary point is a single 1 and so this bit is not stored in memory.

Ex: Convert 2513_{10} to its binary representation using normalized notation and write it in double precision form.

20 Errors with Numbers Stored in Memory

Ex: Describe the difficulty of adding 105_{10} to 84_{10} using two's complement.

Here we have an *overflow error* where a result of a calculation is too large to store in memory.

Ex: How can overflow error be dealt with?

Many real numbers cannot exactly be stored in a computer. For example, $1.01\overline{1011}_2$ cannot be stored in any computer register of finite size. We thus approximate these numbers for storage purposes by *truncating* them (only keeping the most significant figures that we can) or *rounding* them (keeping the most significant figures that we can, increasing the rightmost one if the first discarded figure is at least half of the base).

Ex: Round and truncate each of the following numbers to the required number of significant figures.

1. 1073.21_8, 5 figures

2. $A.492_{16}$, 2 nibbles

3. 10.011010, 4 bits

4. $0.\overline{8}_{10}$, 7 digits

We can also get a *conversion error* when we convert a decimal number to binary for storage and then convert it back to decimal.

Ex: Convert 0.7_{10} to binary and round it off so that it can be stored in single precision form.

Practice: Convert 0.4_{10} to binary and round it off so that it can be stored in single precision form.