# STUDENT LESSON

21 – Number Systems

**INTRODUCTION:** Knowing how the computer stores information will give you an understanding of some of the common computational errors such as over-flow, round-off and, under-flow errors. This lesson is a very light introduction to number systems and their relationship to Computer Science.

The key topics for this lesson are:

A. Introduction to Number Systems

B. Review Decimal

1. Octal
2. Hexadecimal
3. Binary
4. Short Cuts Between Binary, Octal, Hexadecimal
5. How this Relates to Computers and Overflow Errors

**VOCABULARY:** BINARY DECIMAL

HEXADECIMAL OCTAL

OVER-FLOW ERROR ROUND-OFF ERROR

UNDER-FLOW ERROR

**DISCUSSION:** A. Introduction to Number Systems

1. Our number system has 10 as its base. This numbering system (decimal) may have arisen quite naturally from counting on our ten fingers. There are other common number bases that we use without thinking about them, such as when we deal with time (base 60 or sexagesimal): there are, 60 minutes in an hour, and 60 seconds in a minute. Aviators also use a different base when they are dealing with directions and degrees.

2. In Computer Science, we use binary (base 2), octal (base 8), and hexadecimal (base 16) quite a bit. After learning how the bases work, we will look into their relationship to modern computers and how computer calculations are performed.

1. Review Decimal

1. In reviewing our number system, let’s look at what the number 1234 means to us. We are taught very early on about place values. In the decimal number system, starting at the rightmost digit we have the ones place, then the tens place, hundreds place, and finally the thousands place. We are only allowed the digits 0 through 9. The number 1234 represents the following components:

1\*1000 + 2\*100 + 3\*10 + 4\*1

or

1\*103 + 2\*102 + 3\*101 + 4\*100

We are very accustomed to using decimal numbers so we do not think about breaking a number into its components. However, it is important to learn and understand what numbers represent, often by using hands-on activities.

Example: Given an integer variable num, reverse the number. If we knew there were three digits, we could

int a = num/100;

int remainder = num%100;

int b = remainder/10;

int c = remainder%10;

//now we could reassemble it

num = c\*100 + b\*10 + a;

This is similar to what we did in Lab Assignment A3.2, *Coins*.

If we do not know how many digits are in the number, we could use a loop.

int x = 12345;

int newX = 0;

int temp = 0;

while(x > 0){

temp = x%10;

newX = newX \* 10 + temp;

x /= 10;

}

x = temp;

2. If we move to the right of the decimal place, we start using negative exponents.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 103 | 102 | 101 | 100 | 10-1 | 10-2 |

C. Octal

1. After understanding how the decimal number system works, it is quite easy to learn other bases, such as base 8 (octal). They all work basically the same way. In base 8, the available digits are 0 through 7, and each number occupies a place value. The place values are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 83 | 82 | 81 | 80 | 8-1 | 8-2 |

We can designate that a number is in a base other than base 10 (decimal) by using subscripts. Note: If there is no subscript, the number is assumed to be decimal. Let's use the number 123.6 in base 8 – it must be written as 123.68.

The number 123.68 can be converted to a decimal as:

1\*82 + 2\*81 + 3\*80 + 6\*8-1

1\*64 + 2\*8 + 3\*1 + 6\*(1/8)

83.75

Now, using the number 567, we'll show how to convert it from the decimal system to the octal system. Start by looking for the largest power of 8 less than 567. This would be 512 or 83. So in the 83 place value, we put a 1. That leaves us with a remainder of 55. The next place value is 82, but 64 is greater than 55 so that place holds a zero. The next place value is 81. 55 divided by 8 gives 6 for the 81 place value, with a remainder of 7 left over for the next column. The leftover gets placed in the ones column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 83 | 82 | 81 | 80 | 8-1 | 8-2 |
| 1 | 0 | 6 | 7 | 0 | 0 |

or 10678

1. Hexadecimal

1. Hexadecimal is base 16. This base works exactly the same as base 10 and base 8. Hexadecimal will allow the digits 0 through 15. This gets confusing, however, because the digits 10 through 15 are double digits and raises the question: is 10 a single value or two symbols? To avoid confusion we use other symbols, A through F, to represent the values of 10 through 15. Let's use the hexadecimal number 34CD and convert it to decimal. This number has the value of:

3\*163 + 4\*162 + 12\*161 + 13\*160

12288 + 1024 + 192 + 13

13517

So the hexadecimal number 34CD is equivalent to 13517 in base 10 (decimal).

Converting from decimal to hexadecimal is the same process used for octal earlier:

12547 ->

12547/( 163) -> 3, remainder 259

259/( 162) -> 1, remainder 3

3/( 161) -> 0, remainder 3

So 12547 in the decimal number system is equivalent to 310316 in the hexadecimal system.

1. Binary

1. Not surprisingly, binary works the same way as octal and hexadecimal. Binary is base 2, so the only digits that are used are 0 and 1, and the place values are all powers of 2. A binary digit is called a bit. The place values for binary are shown in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | 22 | 21 | 20 | 2-1 | 2-2 |
| 8 | 4 | 2 | 1 | 1/2 | 1/4 |

Let's convert two decimal numbers into the binary system: 13 and 482.  
  
The decimal number 13 -> 11012 (1\*23 + 1\*22 + 0\*21 +1\*20)

The decimal number 482 -> 1111000102

Calculations for converting 482:   
Find the largest power of two that divides into 482. So 28, or 256.

482/256 = 1 , subtract to get the leftover, 226.   
Now check to see if 27, 128 goes into 226.   
Notice we only have the choice of 1 or 0 so this will be simple to calculate. 226 – 128 -> 98 (another 1)

98-64 -> 34 (another 1)

34-32 ->2 (another 1)

can’t subtract 16 (so a 0 goes here)

can’t subtract 8 (another 0)

can’t subtract 4 (another 0)

2 – 2 -> 0 (another 1)

can’t subtract 1 (another 0)

Answer: 1111000102

1. Short Cuts Between Binary, Octal, and Hexadecimal

1. There is a relationship between binary, octal, and hexadecimal numbers. Binary is base 2, octal is base 8(23) and hexadecimal is base 16(24). Let's convert the binary number 101101011010111 to a hexadecimal number.

Binary -> hexadecimal

101101011010111

Starting at the rightmost digit, split the number into groups of 4

101 1010 1101 0111

2. Each of these groups has 4 binary digits that can range from 0 –15. This is exactly the value of one hexadecimal digit, so match each group of four bits with one hexadecimal digit.

|  |  |
| --- | --- |
| **Binary Number Groups** | **Hexadecimal Equivalent** |
| 101 | 5 |
| 1010 | A |
| 1101 | D |
| 0111 | 7 |

So our binary number is equivalent to 5AD716. Going from hexadecimal reverses the procedure so each hexadecimal digit expands to 4 bits.

The same process occurs between octal and binary using only 3 bits.

Try the following conversions for practice:

10 110 1012 -> \_\_\_ \_\_\_\_ \_\_\_\_8

3F116 -> \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_2

3528 -> \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_2

482 ->\_\_\_\_\_\_\_\_\_\_\_\_2

482 -> \_\_\_\_\_\_\_\_\_\_\_8

482 -> \_\_\_\_\_\_\_\_16

100012 -> \_\_\_\_\_\_\_10

57768 -> \_\_\_\_\_\_\_10

3DB16 -> \_\_\_\_\_\_\_10

110 111 0102 -> \_\_\_ \_\_\_\_ \_\_\_\_8

1011 0010 11112 -> \_\_\_ \_\_\_\_ \_\_\_\_16

3FA16 -> \_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_2

7128 -> \_\_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_2

The answers appear at the end of this lesson.

1. How this Relates to Computers and Overflow Errors

1. Modern computers are binary computers. They are made up of switches. Each switch is either on or off at any one time. Everything we put into the computer (numbers and letters) must be converted to binary numbers.

2. At some point in time when using a computer, there are not going to be enough bits to accurately represent the decimal numbers in the real world. We will look at this in a smaller scenario, in order to simplify this discussion.

3. Let’s hypothesize that we have 8 bits for our numbers and there are no negative numbers.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

1. The numbers available to us range from 0000 0000 to 1111 1111 (binary), which is 0 to 255 (decimal). This is not very large for an integer. Attempting to put a larger number into one of our integers would result in an overflow error. Look back at the charts in Lesson A3 for minimum and maximum values for the different types of primitive data types.

5. Another computer problem arises when using numbers that have decimal places. Again we will simplify the real process. We will allocate in our 8-bit storage 6 bits for the *mantissa* (the part to the left of the decimal place) and 2 bits for the fractional portion. We can still have overflow if we try to put a number that is too large for our storage. This storage will hold numbers from 0 to 63.75.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 24 | 23 | 22 | 21 | 20 | 2-1 | 2-2 |
| 32 | 16 | 8 | 4 | 2 | 1 | 1/2 | 1/4 |

1. Let’s convert 7.25 to our binary representation.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 24 | 23 | 22 | 21 | 20 | 2-1 | 2-2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |

Perfect!

Now try 7.4. The first 6 bits are the same, 000111, but what are we going to do for the decimal places? We have four choices:

00 -> 0

01 -> .25

10 -> .5

11 -> .75

We have no other choices. So it looks like we would have to choose between .25 and .5. This is called round-off error. Converting from decimal numbers with fractional parts to a binary representation can cause errors. We are using a very simplified version here, so the errors would usually occur much further out in the decimal places. Most of the time we would not see these errors but as programmers, we should be aware of them. This is why it is never a good idea to compare two floating numbers using “==”. It is much safer to compare the two and check for a certain level of accuracy.

(Math.abs(x – y)< .000000001) is better than (x == y)

This is also why a programmer should choose the best data type for the job.

7. The third type of error is an under-flow error. Let's use our 8-bit storage scheme again, and try to store the decimal number .1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 24 | 23 | 22 | 21 | 20 | 2-1 | 2-2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |

The first 6-bits are 000000. Remember the four choices of decimals:

00 -> 0

01 -> .25

10 -> .5

11 -> .75

The number we are trying to store is too small for our storage scheme. Converting from decimal numbers to binary does not work for numbers very close to 0.

Answers to Practice Questions:

10 110 1012 -> 2658

3F116 -> 0011 1111 00012

3528 -> 011 101 0102

482 -> 0001 1110 00102 (Hint: convert to hexadecimal 1st)

482 -> 7428

482 -> 1E216

100012 -> 1710

57768 -> 307010

3DB16 -> 98710

110 111 0102 -> 6728

1011 0010 11112 -> B2F16

3FA16 -> 0011 1111 10102

7128 -> 111 001 0102

|  |  |
| --- | --- |
| **SUMMARY/ REVIEW:** | There are some dangers and pitfalls in assuming that a computer will always give you the correct answer. Understanding the inner workings of a computer will help you write better programs. Also, hexadecimal numbers are used quite commonly in web design and also in many graphical editing programs. |

**HANDOUTS AND**

**ASSIGNMENTS:** Lab Assignment A21.1, *Numbers*

Worksheet A21.1, *Base Conversions*