CS-160 Lab #3: Hexadecimal and Two's Complement

General

Read through the "Background" material below, then download the Lab #2 question set and answer the questions. Turn in the questions using the instructions posted on the class web site.

For ALL Word processing documents, you must submit your documents in one of the following formats: MS-Word (NOT Works), RTF (most word processors can save in this format), or Open Document (used by the freely available Open Office suite). They will be returned ungraded if submitted in any other format.

Concepts

This lab will give you an introduction to additional representation systems important to computer operations: hexadecimal, and binary two's complement representations (for signed integers).

Background

Note: Original source of the background contained below is from the "CS160 Worksheets" by Daniel Balls of the CS department at Oregon State University; updated and revised by Mitch Fry (CS, Chemeketa Community College).

Hexadecimal Integer representations

In the previous worksheet we worked with a number system that is very important to computer operation—the binary system. Another non-decimal number system that plays an important role in computer science is the hexadecimal system (base 16 numbers). This number system has 16 digits. Since there are only 10 Arabic numerals, we need additional symbols to represent the values 10, 11, 12, 13, 14, and 15. We'll let the letter A represent the value 10, B represent the value 11, and so on, through F, which will represent the value 15.

An example of a hexadecimal number is 4C7. To compute the decimal value of this number, it is important to know that each place value is associated with a power of 16 in the same way the powers of ten are associated with the decimal system and the powers of two are associated with the binary system. So 4C7 is equivalent to the following decimal number:

$$4\cdot 16^2 + \mathrm{C}\cdot 16^1 + 7\cdot 16^0 = 4\cdot 256 + 12\cdot 16 + 7\cdot 1 = 1024 + 192 + 7 = 1223.$$

Another number system that has application in the field of computer science is the octal number (base 8) system. This system has 8 digits—0, 1, 2, 3, 4, 5, 6, and 7. As its positional values, the octal system uses the powers of 8 (but you may have guessed that by now). Then 452 in the octal number system is equivalent to the decimal number:

$$4\cdot 8^2 + 5\cdot 8^1 + 2\cdot 8^0 = 4\cdot 64 + 5\cdot 8 + 2\cdot 1 = 256 + 40 + 2 = 298.$$

We will not study octal numbers any further for this class (nor will they appear on exams) as they are used less frequently than binary and hexadecimal representations.

The next activity will make you more familiar with hexadecimal numbers.

Lab #3: Hexadecimal and Two's Complement

Practice Exercise: Fill in the blanks with the appropriate hexadecimal and octal integers.

Decimal	Hex.	Decimal	Hex.
1		51	
2		52	
3		53	
4		54	
5		55	
6		56	
7		57	
8		58	
9		59	
10	**A**	60	
11		61	**3D**
12		62	
13		63	
14		64	
15		65	
16		66	
17		67	
18		68	
19		69	
20		70	
21		71	
22	**16**	72	
23		73	
24		74	
25		75	
26		76	
27		77	
28		78	
29		79	
30		80	
31		81	
32		82	
33	**21**	83	
34		84	
35		85	
36		86	
37		87	
38		88	
39		89	**59**
40		90	
41		91	
42		92	
43		93	
44		94	
45		95	
46		96	
47		97	
48		98	
49		99	
50		100	

Hexadecimal and Decimal Conversions

The procedure for converting numbers between the hexadecimal and decimal number systems should have a familiar feel: to convert a decimal number to its hexadecimal equivalent, first find the decimal number's binary form, then arrange the bits from right to left into groups of four. For example, to find the hexadecimal equivalent of the decimal 269, we first need to obtain the binary form of 269. Using either of the methods present in the last worksheet, it can be shown that the binary equivalent of 269 is 100001101. Regrouping this binary form into strings of 4 bits (and adding three leading zeros on the LEFT) shows that the decimal 269 can be written as 0001 0000 1101. These three strings correspond to the hexadecimal 1, 0, and D, respectively. Therefore the hexadecimal equivalent of the decimal 269 is 10D.

To find the decimal equivalent of the hexadecimal integer 82C, we'll first write this number in a binary form: 1000 0010 1100 = 100000101100. Next, we add appropriate powers of two to obtain the result: $1 \cdot 2^{11} + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 = 2048 + 32 + 8 + 4 = 2092$. Therefore the hexadecimal number 82C has an equivalent decimal form of 2092.

Two's Complement

Up to this point we have seen how it is possible to represent whole numbers (e.g. 0, 1, 2, 3, …) using bits. But can one represent the integers (i.e., the counting numbers and their negative counterparts) using the binary system? The answer to this question is yes. The most common system for representing the integers is called *two's complement* notation. Although most computers today use a 32 bits to represent the integers, learning the two's complement system with that many bits would be too cumbersome. Therefore, we will study the features of the two's complement system using only 6 bits. As we shall soon see, using a 6-bit pattern, we are able to represent all the integers from -32 to 31. It is important to note that the 6-bit pattern and the 32-bit pattern are identical in structure, so studying the 6-bit pattern does have merit. The 32-bit pattern is used in actual computers so that more integers (from -2,147,483,648 to 2,147,483,647) can be represented. The table below contains some of the representations of the integers in a 6-bit two's complement system.

Value Represented	Bit Pattern
31	011111
30	011110
…	…
5	000101
4	000100
3	000011
2	000010
1	000001
0	000000
–1	111111
–2	111110
–3	111101
–4	111100
–5	111011
…	…
–30	100010
–31	100001
–32	100000

In any two's complement system, the leftmost bit represents the sign (+ or –) of the integer. For this reason, it is called the *sign bit*. The integer is negative when the sign bit is 1; the integer is either zero or positive when the sign bit is 0. To represent the positive integers leading zeros are added as needed to the left of the normal binary representation of the integer. For example the number 23, which has a binary representation 10111, is represented as 010111 in a 6-bit two's complement system. Using a 6-bit two's complement system, if the uppermost (leftmost) bit is one, then the number is a negative value; the uppermost bit would represent the value -2^5 (-32). The uppermost bit for a 7-bit two's complement system represents the value -2^6 or -64. For example, the number 1011001 can be converted to a decimal representation as follows:

$$1\cdot(-2^6) + 0\cdot2^5 + 1\cdot2^4 + 1\cdot2^3 + 0\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = (-64) + 16 + 8 + 1 = (-39)$$

The easiest way to find the representation of a negative integer is to begin with its positive counterpart. For example, to find the two's complement representation for the integer –23, we start with the two's complement representation of 23: 010111. At this point there are several methods that can be used to determine the 6-bit two's complement representation of –23. Both methods are described below.

The first method is referred to as the 'copy and complement' method. Beginning at the right, the bits of the positive integer are copied up to and including the first 1 bit. In our example, since the rightmost digit is 1, it is the only digit copied: **__ __ __ __ __ _1_**. The remaining digits of the negative integer are *the complement* of the remaining bits of the positive integer. The complement of a pattern is found by changing all 0s to 1s and all 1s to 0s. Thus, the complement of the remaining portion of 23 is **1 0 1 0 0** 1, and the integer –23 is written 101001 in two's complement notation

Another method to find the negation of a number in two's complement notation could be named the 'complement and add 1' method. We obtain the complement of the original number and add one to it. Using the same example of 23, we begin by determining the complement of 010111, which is 101000. Adding 1 to this complement yields the same result as before: 101001. Notice that regardless of what method is used, a positive integer—which always begins with 0—will always have an opposite or negation that begins with a 1—and is therefore negative.

Practice Exercise: Using either of the methods described above, find the negation of each of the following numbers, represented in 6-bit two's complement notation:

001110 negation = _____

010010 negation = _____

110011 negation = _____

000000 negation = _____

The ability to negate a number written in two's complement form is crucial in finding the value of any number written in a two's complement system—for example what integer does the number 110110 represent? The first task is to determine whether the integer is positive or negative. This factor depends on the sign bit—the first bit of the string. If the first bit is 0, finding the value of the number amounts to converting the number—which is basically in binary form—into its decimal equivalent. If however the number's sign bit is 1 (and therefore the

number is negative), treating the number as if it is part of the binary system will not work. Instead, we need to find the magnitude of the number and then place a negative sign in front of this magnitude. And what is the magnitude of a negative number? It is that number's *negation*. In our example, the negation of 110110 is 001010 and this number corresponds to the binary number 1010. This binary number has a decimal equivalent of 10, which means 110110 is equivalent to the negation of 10, or –10.

Practice Exercise: Convert these 6-bit, two's complement integers into decimal form.

101000 _____

110011 _____

111111 _____

Practice Exercise: Convert these decimal integers to 6-bit, two's complement form.

14 _____

–27 _____

–11 _____

Binary and Two's Complement Addition

One method of verifying that 101001 represents –23 is to *add* this string to the two's complement representation of 23 and check the result. If the sum is 0, which is the sum of 23 and –23, then we can be sure that 101001 is indeed the representation of –23.

Adding numbers written in two's complement notation is equivalent to adding numbers in the binary system. To add two binary numbers, one must be familiar with some basic base-2 addition facts, much like we must understand basic base-10 addition facts. Fortunately, there are only a few facts to know!

0 + 0 = 0 0 + 1 = 1 1 + 0 = 1 1 + 1 = 10 10 + 1 = 11

The base-10 addition practice of 'carrying a leftover' is used when adding binary numbers. Thus if the binary number 110 is added to the number 111, the result is 1101 as expected and is described below.

```
  1 1 0
+ 1 1 1
-------
  0 1
```

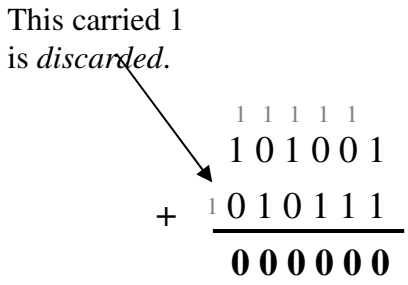
```
    1
  1 1 0
+ 1 1 1
-------
1 1 0 1
```

The 1 and 1 in the middle column add to 10 and the 1 (from 10) is carried to the leftmost column.

The 1 and 1 in the leftmost column, combined with the 1 carried from the middle column, add to 11. The leading 1 (from 11) is carried once again.

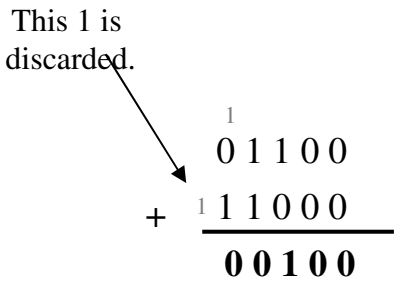
This method is analogous to addition in the decimal system. However, when adding numbers in two's complement notation, a change in the familiar method is necessary since the *number of bits of the sum **must be equal** to the number of bits in each of the addends*. In other words, when two numbers that are represented in a 6-bit two's complement system are added together, the result must also have exactly six bits. What happens when there is an extra 1 that, according to our decimal rule, should be put into a seventh bit position? The answer is that *it is simply discarded*. (This may be counterintuitive, but it works!)

Using our knowledge of addition in the two's complement system, we can now compute the sum of 101001 and 010111, which as shown below does equal 0 (or in 6-bit two's complement notation 000000).



The sum of 101001 and 010111 is 00000, which means they are negations of each other.

One of the powerful features of a two's complement system is that it contains the structure to not only perform additions, but also subtractions. At the foundation of this process is the fact that any subtraction problem can be rewritten as an addition problem [for example, 12 – 8 = 12 + (–8)]. So if a computer has the ability to perform addition (i.e., if it has an adder) it will also be able to perform subtraction. The subtraction problem 12 – 8 could be performed by writing the integers 12 and –8 in two's complement notation and then adding them. This addition is shown below. Note that while we could have used a 6-bit (or 32-bit) system, a simpler 5-bit system is sufficient, since the positive integer is less than or equal to 15 and the negative integer is greater than or equal to -16.



A model of the subtraction problem 12 – 8 using two's complement notation. The number 01100 represents 12 and 11000 represents –8. The expected result 4 will is evident after the leftmost 'carried 1' (created by the addition of 1 and 1 in the leftmost column) is discarded.

Practice Exercise: Using a 6-bit two's complement system, perform the following addition and subtraction problems:

23 + 7 –13 – 18 28 – 19 16 + 20

Did you run into a problem with the last sum (16 + 20)? If you did everything correctly, your sum should have been 100100. But this doesn't make sense, since the sum of two positive numbers is positive. The integer 100100 begins with a one and is therefore negative. The solution to this conflict has to do with a phenomenon called *overflow*. This problem is due to the fact that the integer 36, which is the true sum of 16 and 20, cannot be represented in a 6-bit system. Whenever the sum of two numbers cannot be represented in the current two's complement system, overflow occurs. The problem of overflow is serious because, unless the computer reports the error, the computation could return an erroneous result.

An applet that will help you become more familiar with the concept of overflow is found at: http://newterra.chemeketa.edu/faculty/mfry5/cs160/applets/TwosComplement.html

1. Make sure the option *Demo Mode* is selected.
2. Click on the individual bits to change them from 1 to 0 and 0 to 1.
3. See if you can get the applet to achieve overflow. The circle above the word *Overflow* will turn red when overflow occurs.
4. By clicking on the arrow to the right of number 4 you can change the number of bits in the system to 6 or 8.