Nybbles and Floats

Due Friday, March 7\(^1\) (Corrected March 3, 2003)

1. This is an exercise on writing simple bit manipulations in assembler. You will be working with nybbles, which are 4 bits or 1 hexadecimal digit. Thus cafebabe, the magic number at the beginning of Java's .class file, is 8 nybbles. Java's 0x1234 is 4 nybbles since this is the hexadecimal 1234. Since we are not implementing objects (yet), we'll keep nybbles in an int. Thus an integer can contain 8 nybbles; nybble 0 is the rightmost nybble and nybble 7 is leftmost. For example, the 5\(^{th}\) nybble of 0xcafebabe is 0xf, or 15.

Specifications. You will write the following static methods in the class Nybbles.

(1) public static int get(int x, int nybbleNum) will return the value of x's nybbleNum\(^{th}\) nybble. You may assume that nybbleNum contains a value from 0 to 7. For example, get(0xcafebabe, 5) returns 15 (0xf).

You should p-code before coding; otherwise you'll be fumbling around. Look at paragraph 20 in my part 2 notes which shows you how to get a particular bit. Now you want to get a particular 4 bits. So if you need nybble 3, you'll need to shift (insert correct number) bits. How do you get (insert correct number) from 3? Your code will do this more efficiently; I do not want to see imul.

(2) public static int clear(int x, int nybbleNum) will "clear" x's nybbleNum\(^{th}\) nybble and return that value. You may assume that nybbleNum contains a value from 0 to 7. For example clear(0xcabebabe, 2) returns 0xcabeb0be, that is, -889,278,274.

Here you want to set 4 bits to 0. Paragraph 18 shows how to set one bit to 0. So your mask will be four 0's rather than one. So start out with 1111\(_2\) rather than 1\(_2\).

(3) public static int set(int x, int nybbleNum, int nybble) will "set "

---

\(^1\) The last day before vacation. But I will not look at them before March 12. So if you are not going to Florida, you’ll have more time.
x's nybbleNum-th nybble to nybble and return that value. For example set(0xcafebabe, 5, 13) will return 0xcafebabe, that is -891,372,866. To accomplish this you will first use clear(x, nybbleNum) to clean out the old value before replacing it with the new. (I want you to call another method from within a method.)

After clearing out the nybble, setting it to nybbleNum should be easy.

Testing. In a separate class (choose your own name), write Java code to test these methods. In particular, let x = -2,023,406,815. Show the values of get(x,i), clear(x,i), and set(x,i,10), where i runs from 7 down to 0. You should get nice answers.

This is one way of displaying the results (you can, of course, do it your way):

C:\Java\jasmin>java TestNybbles 4099
x in hex is 1003
get(x,7) = 0x0   clear(x,7) = 0x1003   set(x,7,10) = 0xa0001003
get(x,6) = 0x0   clear(x,6) = 0x1003   set(x,6,10) = 0xa001003
get(x,5) = 0x0   clear(x,5) = 0x1003   set(x,5,10) = 0xa01003
get(x,4) = 0x0   clear(x,4) = 0x1003   set(x,4,10) = 0xa1003
get(x,3) = 0x1   clear(x,3) = 0x3   set(x,3,10) = 0xa003
get(x,2) = 0x0   clear(x,2) = 0x1003   set(x,2,10) = 0x1a03
get(x,1) = 0x0   clear(x,1) = 0x1003   set(x,1,10) = 0x10a3
get(x,0) = 0x3   clear(x,0) = 0x1000   set(x,0,10) = 0x100a

Programming note. The method Integer.toHexString(x) will allow you to see the value of an integer in hexadecimal. Use it.

2. This is an exercise on writing simple floating point manipulations in assembler.

Specification. In the class FloatFunctions, to be written in jasmin, you will define and code the following method:

public static float geometricMean(float a, float b)

which computes geometric mean of a and b. Please site the source for your formula.

In a separate class (choose your own name), write Java code to test this method. In particular, show the values of geometricMean(32.0f, 2.0f) and geometricMean(10.0f, 20.0f). (Note the f's. 1.3 in Java is double.)

For example, my output looks like:

2 The geometric mean answers the question: If you have a rectangle with sides 32 and 2, then what is the side of a square with the same area? (8)
C:\Java>java TestFloat 32.0 2.0
The geometric mean of 32.0 and 2.0 is 8.0

Programming notes:

(1) Floats are a distinct data type in most architectures. Think of a float in scientific notation, such as -3.14*10^2 that is somehow written in binary. I've included notes on floats below if you are interested. Since they are different, the JVM and other architectures will have separate instructions for them. The JVM's are quite simple. Your job is to find them and then use them. The data type for specifying a float parameter is F.

(2) You will need to call the square root function in the Math class. The function expects double (64 bits), not float and returns a double, not float. What to do, then? The JVM has conversion instructions f2d and d2f. Look them up and use them. The data type for specifying a double parameter is D.

(3) I used Float.parseFloat(args[0]) to read floats from the command line. Or you can use Keyboard.readFloat() to read from the console. Of course, you don't have to do this at all.

What to do and other things

1. This must be an electronic submission. I want just one script file containing both parts. Each part will have source, compiles, and tests similar to what you did before. I need to easily read what you are trying to do.

2. You must follow the style that I developed for writing jasmin programs:
   Java method definition and short description
   Parameters and local variable descriptions.
   P-code. Required. Pseudo code should be of sufficient detail that a good programmer could write code for it in any language. Therefore, nothing in your pseudo code should be Assembler specific. Right now it's just simple expressions.
   Stack contents after each instruction.
   Comments describing the purpose of the code. Comments clarify (not echo) instructions.

   Java code should be pleasant to read. It need not be fancy.

3. You can and should work together for the analysis and design (which includes p-code) of the program. You must do your own code and commenting. Plagiarism has dire consequences.
1. This must be an electronic submission. I want just one script file containing both parts. Each part will have source, compiles, and tests similar to what you did before. I need to easily read what you are trying to do.

2. You **must** follow the style that I developed for writing jasmin programs:
   - Java definition and short description
   - Parameters and local variable descriptions.
   - P-code. **Required.** Pseudo code should be of sufficient detail that a good programmer could write code for it in any language. Therefore, nothing in your pseudo code should be Assembler specific. Right now it's just simple expressions.
   - Stack contents after each instruction.
   - Comments to clarify (not echo) instructions.

   Java code should be pleasant to read. It need not be fancy.
Floating point numbers.

Look in Tanenbaum pp. 565-573

1. Think scientific notation: \(-1.5 = -1.5e0\). We need a sign, an exponent and a mantissa (fraction).
   Of course we should do this in binary, so \(1.1_2e0\). \(e0\) means times \(2^0\). To the right of the decimal point is
   halves, quarters, eighths, etc. \(\frac{1}{4} = 0.01 = 1.0e-2\) \(\frac{1}{8} = 0.101 = 1.01e-1\) \(13.0 = 1101 = 1.101e3\)

2. Floating point numbers take up storage. Normally float is 32 bits; double is 64 bits. We’ll do the IEEE
   standard for float. You can look at the table on p. 571 for double.

<table>
<thead>
<tr>
<th>bits: 1</th>
<th>8</th>
<th>223</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exp = e + 127</td>
<td>fraction: 1.xxx xxxx xxxx xxxx xxxx xxxx (note implied 1.)</td>
</tr>
</tbody>
</table>

   IEEE tries to have the numbers make sense; thus the concept of a normalized number. We must have
   \(0 < \exp < 255\), that is \(-127 < e < 128\).

   \(\exp = 0\) and fraction = 0 means 0 (we can have \(\pm 0\)).
   \(\exp = 255\) and fraction = 0 means \(\pm \infty\)

   Now comes the special cases:

   \(\exp \exp = 0\) and fraction \(\neq 0\) means a denormalized number: we have
   \(0.xxx xxxx xxxx xxxx xxxx xxxx\)
   See the implicit 0. This allows smaller numbers at the cost of accuracy.

   \(\exp = 255\) and fraction \(\neq 0\) means NAN (not a number)

3. What is the largest? \(1.11111\ldots e127 = 2.0e127 = 1e128 \approx 3.40e38\)
   What is the smallest normalized? \(1.000\ldots e-126 \approx 1.17e-38\)
   What is the smallest denormalized? \(0.0000\ldots e-126 = 1.0e(-126 - 23) \approx 1.40e-45\)

4. Write 15.0 as float. \(15.0 = 1111.0 = 1.111e3\); exp = 130 = 1000 0010

   So 0 1000 0010 1110 0000... In hex: 41700000

   Write \(\frac{1}{3}\) as a float: Double. If \(\geq 1\) write a 1 and subtract 1; else 0. Repeat.
   \(\frac{1}{3}*2 = \frac{2}{3} < 1\). Write 0
   \(\frac{2}{3}*2 = 4/3 \geq 1\). Write 1 and subtract 1 to get \(\frac{1}{3}\).

   So get .01010101 \(\ldots\) So have 1.010101\ldots e-2; exp = 125 = 0111 1101

   So 0 0111 1101 0101 0101 0101 0101 0101 011 (rounding) = 3EAAAAAB
5. It should be clear that adding floating point numbers is a lot different than adding integers. Thus most machines have a separate floating point unit. And that means another architecture.

(Material for the SPARC)

6. 32 floating point registers %f0 - %f31. (double uses a pair of registers. quad uses four registers.) That’s it! What this means that if a function using floating point registers calls another subroutine, there is no guarantee that the floating point registers will have the same values after the call. We may need to save floating point numbers on the stack before making a call.

7. There is also the FPU Control/Status register (FSU) which contains the floating point condition codes (fcc). Bits are set after a floating point compare for operands =, <, >, ? (compare 1.0 with NAN)

8. We need separate instructions to load (LDF) and store (STF) floating point registers. The assembler will know that ld %f0 means LDF.

Note: the FPU and CPU can be on different chips. There is no way (yet) of moving data directly from a general register to floating point register. The data must be first saved to memory (usually the stack) and then loaded.

There are separate floating point instructions (I’m looking at SPARC 8, not SPARC 9):

```
fadds fsubs
fmuls fdivs arithmetic
fcmps compare
fbx branch on x, such as greater, less....
fitos integer to single. Conversion
fstoi single to integer. Conversion
fstod single to double....
fmovs move. No register containing 0.0 so we cannot use fadds
fneg negate
fasbs absolute
fqrts square root
```

9. Passing float parameters: in the general purpose registers! The subroutine must store them in memory and then load them into floating point registers.

Returning a float: in %f0.