ICSI 404 – Assignment 2, the ubiquitous longword

This assignment builds on the previous assignment. A single bit, by itself, is not very useful. In order to represent values and addresses, we need multiple bits. For the machine that we are simulating, we will be using a 32-bit value for both addresses and values. A word is a collection of bits, originally the size that the machine “natively” works on. Along the history of computing, a word became 16 bits. A longword is a 32-bit collection of bits.

You must fully implement this interface (source file is provided). You must make a new class called “Longword” that does not inherit from anything. You **must** create a collection (array is best) of Bit (from assignment 1) and use that for storage. You **may not** use any other storage mechanism.

```
public interface ILongword {
    bit getBit(int i); // Get bit i
    void setBit(int i, bit value); // set bit i's value
    longword and(longword other); // and two longwords, returning a third
    longword or(longword other); // or two longwords, returning a third
    longword xor(longword other);// xor two longwords, returning a third
    longword not(); // negate this longword, creating another
    longword rightShift(int amount); // rightshift this longword by amount bits, creating a new longword
    longword leftShift(int amount);// leftshift this longword by amount bits, creating a new longword
    @Override
    String toString(); // returns a comma separated string of 0's and 1's: "0,0,0,0,0 (etcetera)" for example
    long getUnsigned(); // returns the value of this longword as a long
    int getSigned(); // returns the value of this longword as an int
    void copy(longword other); // copies the values of the bits from another longword into this one
    void set(int value); // set the value of the bits of this longword (used for tests)
}
```

Unlike what is actually done in hardware, you may use loops to implement the same operation done on each bit. You must use the operations from bit (and, or, not, xor, getBit, set) where appropriate. You must validate inputs where appropriate.

For getSigned() and getUnsigned(), the wrong way is to convert to a string, then call Integer.parseInt(). Do not do this. Instead, use the powers of two: the rightmost bit is worth 0 or 1. The next bit is worth 0

or 2; the next is 0 or 54. You could use Math.Pow() for this, but since it is always multiplying by 2, I would just keep a "factor" and multiply by 2.

Consider (by hand), we will convert 1010 to integer:

rightmost is 0. 0 * 1 is 0. Add that to a sum (0). Factor is 1*2 = 2.

next is 1. 1 * 2 is 2. Add that to the sum (2). Factor is 2*2 = 4.

next is 0. 0 * 4 is 0. Add that to the sum (2). Factor is 4*2 = 8.

next (and last) is 1. 1 * 8 is 8. Add that to the sum (10). Factor is 8*2 = 16.

Answer is sum (10).

Set should use the same logic, but backwards. Given an integer, you should perform the algorithm from getSigned() **backwards** to determine which bits should be set to make the longword have the input value.

You must provide a test file (longword_test.java) that implements void runTests() and call it from your main, along with your bit_test.runTests(). As with the bit test, these tests must be independent of each other and there must be reasonable coverage. You can not reasonably test all 4 billion possible longwords, but you can test a few representative samples.

You must submit buildable .java files for credit.

Rubric	Poor	OK	Good	Great
Comments	None/Excessive (0)	"What" not "Why", few (5)	Some "what" comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Unit Tests	None (0)	Partial Coverage (7)	All methods covered, needs more cases (13)	All methods/cases covered (20)
Accessors/ Mutators /toString	None (0)	Some implemented (5)	All implemented, some fail (7)	All implemented, all tests pass (10)
And	None(0)		Implemented, wrong (2)	Implemented, correct (5)
Or	None(0)		Implemented, wrong (2)	Implemented, correct (5)
Not	None(0)		Implemented, wrong (2)	Implemented, correct (5)
Xor	None(0)		Implemented, wrong (2)	Implemented, correct (5)
Left/Right shift	None(0)		Implemented, wrong (5)	Implemented, correct (10)
GetSigned/Unsigned	None(0)		Implemented, wrong	Implemented, correct

			(5)	(10)
Copy/Set	None(0)		Implemented, wrong (5)	Implemented, correct (10)