ICSI 404 – Assignment 1: The Bit and the Word

In this series of programming assignments, you would go through the step-by-step design process of a simple computer (along with its Instruction Set Architecture) and develop a software emulator for this machine. You would then complete the series by developing an assembler that can assemble a simple program written in using the instruction set to machine code, which is executed by the emulator.

For the first assignment, your job is set up the basis for internal data representation in the CPU, namely the bits and the word. While Java provides bit-manipulation capabilities using bitwise operators (&, |) and bit-shifts (<<,>>), those bit manipulations are to be done within some form of an integer (byte, short, int or long). However, Java's lack of support for unsigned integer type comes as an impediment to having full control for our own algorithms for bit-level arithmetic implemented. The next best option is to use the "boolean" data type, where we can use the "true" value for a "1" and "false" value for a "0". (Internally, Java uses a whole byte for storing these two possible values, because a byte is the smallest storage allocation possible for storing a variable. One can also use a byte-sized integer, which is going to take the same space, but the handling the exception when any value other than 0 or 1 is set becomes the programmer's burden.)

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 emulating, we will be using a 32-bit value for both addresses and values. A word is an indexed collection of bits (like an array), originally fitting the size that the machine "natively" works on. Along the history of computing, a word became 16 bits. A 32-bit collection of bits was then named as "long-word" (Similarly, 64-bit is "quad-word"). The `BitSet` class in `java.util` package, designed for representing a set using a bit-vector, can be repurposed for our need. It is much more space-efficient in that it uses only one bit internally for each bit/boolean value stored in an array of long (64-bit) integers. To satisfy our need for 32 bit/boolean values, just a single-element array suffices as that single long integer can pack up to a maximum of 64 bit/boolean values. Moreover, most of the operations we need to perform on our machine's long-word are easily implemented using the built-in methods of the `BitSet` class, often using simple wrappers.

For our 32-bit machine word, the `LongWord` class implementation **must follow these specifications**:

1. The `LongWord` class must be **a new class that does not inherit from (i.e., extend) any other class** (i.e., it implicitly extends only the Java `Object` class).
2. A LongWord object **must** have a bit-vector (indexed collection of bits) as part of its data field, to be implemented using the Java **`BitSet`** class, storing the 32 bits of the long-word. Treat the 0th bit of the bit-vector as the least significant and the 31st bit as the most significant bits. For the sake of preserving consistency across the class, you **may not** use any other storage mechanism. Do not forget to import `java.util.BitSet` in your source code.
3. You **must** create the API for your class by implementing the following methods:
   a) At creation, every bit of a LongWord object must be initialized to "0". Follow the BitSet class documentation to figure out anything you need to do in the constructor to ensure this.
   b) Override default `toString` method of Java objects, also being careful not to make use of the default `toString` method of the `BitSet` data field, which only lists the bit-indexes (for which

the boolean/bit value is true/1) in the set notation. Instead, the printable string form should have all the 32 bits listed as 0/1 with the most significant bit in the leftmost position and the least significant bit placed at the rightmost. You may like to group them by 4 or 8 bits with a space in between. This should be followed by a tab ('\t') as delimiter and the equivalent 8 hex digits for the binary pattern prefixed by 0x, indicating that what follows are hex digits. A typical string representation may look like the following:

```
1010 1010 1010 1010 1010 1010 1010 1010   0xaaaaaaaa
```

You **must use `@Override` annotation** to make the compiler enforce overriding of `toString`.

```
@Override
String toString(); // returns a string of 0's and 1's
                        // followed by equivalent hex notation
```

c) Implement the following public **bitwise accessors and mutators**:

```
boolean getBit(int i) // Get bit i as a boolean
void setBit(int i) // set bit i (set to true/1)
void clearBit(int i) // clear bit i (reset to false/0)
void toggleBit(int i) // toggles (flips) bit i
```

d) Implement the following public **accessors and mutators for the whole long-word**:

```
int getSigned(); // returns the value of this long-word as an int
long getUnsigned(); // returns the value of this long-word as a long
void set(int value); // set the value of the bits of this long-word
                  // (used for testing only)
void copy(LongWord other) // copies the values of the bits from
                  // another long-word into this one
```

Note that the bit-pattern representing the largest unsigned 32-bit integer has a 1 in the most significant 31st bit and Java will interpret it as negative, if the integer is left as an `int`. Thus we need a `long` as the return type for `getUnsigned`.

e) Implement the following public methods for bit shift operations:

```
LongWord shiftLeftLogical(int amount); // left-shift this long-word by
            // amount bits (padding with 0's), creates a new long-word
LongWord shiftRightLogical(int amount); // right-shift this long-word
            // by amount bits (padding with 0's), creates new long-word
LongWord shiftRightArithmetic(int amount);// right-shift this long-word
            // by amount bits (sign-extending), creates a new long-word
```

f) Implement the following public methods for bitwise logical operations:

```
LongWord not(); // negate this LongWord, creating another
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
```

Note that all four of these operations are on the corresponding bit-vector field in the LongWord, and the `BitSet` class implementing the bit-vector already has them implemented as its methods. You are **free to decide** whether to use them or not.

g) Implement the following public method to check if a long-word has all 0-bits:

```
boolean isZero() // returns true if all bits are 0's in this long-word
```

Unlike what is actually done in hardware, you may use loops to implement the same operation done on each bit. You may extract the bits as boolean using `getBit` and invoke normal boolean operators or use `BitSet` build-in methods to operate on multiple bits where appropriate. You **must validate inputs** where appropriate.

For `getSigned` and `getUnsigned`, the wrong way is to convert to a string and then call the method `Integer.parseInt` on it. Do not take this route. 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 4, and so on. You could use Math.Pow() for this, but since it is always multiplying by 2, it is easy to keep a "factor" and iteratively multiply by 2 or shift bits.

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).

Setting bits should use the same logic, but backwards. Given an integer, you should perform the above algorithm **backwards** to determine which bits should be set to make the longword have the input value.

You can also start from the leftmost bit. Think about how to proceed!

Another **caution** if you intend to convert a `BitSet` object to `long` using `toLongArray` and have later plans to re-convert to `BitSet` using `valueOf`. This may have some very unexpected program behavior with exceptions being thrown, especially when the long-word contain an all-zero bit-vector. Because BitSet is designed for representing a set, this implies an empty set with zero cardinality. Consequently, many iteration over bits keep failing, causing the exceptions.

You must provide a test file (`TestLongWord.java`) that implements a void `runTests` method. Each method of the LongWord class must be tested in `TestLongWord.java`. These tests can throw an exception on failure, print expected and actual values or print “pass” or “fail”, for example. The tests must be adequate to prove that your `LongWord` class really works. These tests must be independent of each other, i.e., you do not want one failed test to cause another failed test to occur. Also, there must be reasonable coverage. You cannot possibly test all 4 billion possible long-words, but you can test a few representative samples, including special ones such as 0, -1, largest magnitude signed and unsigned integers, etc. Your main method should call `runTests` on `TestLongWord`. There should be output that allows a TA/grader to determine that the tests pass or fail. They may also have a battery of tests that they will run.

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