LAB A
Translating Data to Binary

Create a directory for this lab and perform in it the following groups of tasks:

LabA1.java

1. Write the Java app LabA1 that takes an `int` via a command-line argument `args[0]` and passes it to the static method **`toBinaryString(int)`** of the `Integer` class. Have your app output the returned string. Assume, as a precondition, that `args[0]` parses to an `int` without exceptions.

2. Compile your program and then run it with argument `5`; i.e. issue the command:

   ```
   java LabA1 5
   ```

   The output is a string of three *bits* (3b). A bit is a binary digit and can be either 0 or 1. A string of bits is known as a *binary pattern*.

3. Run the app again with argument `212`. The output now consists of eight bits and since 8 bits form a *byte*, we can say the output is made up of one byte or 1B.

4. The bits in a binary pattern are numbered right-to-left starting from 0. Bit number 0 in the binary pattern `11010100` is zero while bit number 2 is one. Bit number 0 is called the *least significant* bit (LSb) while the leftmost bit is the *most significant* bit (MSb).

5. Java represents integers in bits using a positional system in which each position is associated with a weight. The weight of bit number k is 2^k. Hence, `11010100` is thus (right to left): $0x2^0+0x2^1+1x2^2+0x2^3+1x2^4+0x2^5+1x2^6+1x2^7 = 4+16+64+128 = 212$.

6. Use the above observation to predict the output of your app when the argument is `24`. Run the app and verify your prediction. Report your results.

7. The positional nature of the binary representation implies that bit number 0 of any even integer must be zero. Verify by trying several even arguments and report them.

8. If the binary representation of an integer ends (at the least significant end) with `00`, what can you conclude about the value of the integer? Use your app to verify.

9. If the binary representation of an integer ends (at the least significant end) with `01`, what can you conclude about the value of the integer? Use your app to verify.
   Hint: These two bits would be dropped if you divided the integer by 4.

LabA2.java

10. Writing down long binary patterns is tedious and error prone. For this reason, it is often preferable to use the hexadecimal notation: partition the bits of the pattern into groups of four (adding zeros on the left if needed) and replace each group by one hexadecimal character using the following table:

```
0000=0, 0001=1, 0010=2, 0011=3, 0100=4, 0101=5, 0110=6, 0111=7
1000=8, 1001=9, 1010=A, 1011=B, 1100=C, 1101=D, 1110=E, 1111=F
```

    For example, the pattern `101101` is written as `2D` in hex. And to indicate that this is hex, the prefix `0x` (C, Java, and assembly) or `'h` (Verilog) is used

11. Save your LabA1 as LabA2.java and modify it so it prints the hex representation of the argument. You do this by using the following method in the `Integer` class:

```
public static String toHexString(int)
```

12. Compile your app and run it supplying `205`. The output should be `0xCD`. It does not matter if the hex digits are written in lower or upper case.

13. Note that Java sets aside 32 bits (4 bytes) to represent an `int`. The higher bits in the above examples were suppressed because they were zeros. To expose this, we will use a method (also in the `Integer` class) that reverses (inverts) the binary pattern:

```
public static int reverse(int)
```

14. Modify your app so it also outputs (in hex) the reverse representation of the argument. Run your app with argument 195. What is the output?

15. Note that if the reversal is done at the byte level (rather than the bit level) then the answer would be different. Such a reversal (for the 32-bit, or 4-byte size) would turn the pattern ABCD into DCBA. Search for a method that performs such a reversal and modify your app so it outputs this result in addition to the earlier two.

16. Run your app with argument 195. What is the output?

LabA3.java

17. Revert back to LabA1 and save it as LabA3. Modify the app so it takes *two* integers via `args[0]` and `args[1]`. Call the corresponding integers `x` and `y` (we continue to assume that the arguments are valid). Your app should output the values of `x` and `y` in binary along with the value of `z` for each of the following four cases:

```
int z = x & y   // and
int z = x | y   // or
int z = x ^ y   // xor
int z = ~x      // not
```

    These are Java's bitwise operators; i.e. they operate at the bit level.

18. Make sure you predict the output of this app *before* you run it to ensure you understand the underlying operation. What are the predicted outputs if x=205 and y=38?

19. Compile your app and run it. Report your results including x, y, and outputs.

LabA4.java

20. Revert back to LabA1 and save it as LabA4. Modify the app so that after it parses the integer `x`, it computes and outputs (in binary) `x` and `z` for each of the following cases:

```
int z = x << 1    // logical left shift
int z = x >>> 1   // logical right shift
int z = x >> 1    // arithmetic right shift
```

    These are Java's shift operators. Note that there are two different types of right shifts but only one type of left shift.

21. Run the program and supply any positive integer as input. Examine the outputs and verify that these shift operators do indeed shift the binary pattern left and right. You will not see a difference between logical and arithmetic right shifts at this stage. Report your results.

22. Modify your app so it outputs the shifted values in decimal as well as binary. Run it several times using a variety of positive arguments and try to discover the arithmetic computation that these shifts perform. Report your results.

23. Modify your app one last time by changing the shift amount from 1 to 2; e.g. the first computation becomes **`x << 2`**.

24. Report the binary outputs and verify that they correspond to shifts by two positions.

25. Examine the decimal outputs and generalize the arithmetic interpretation of shifts. Report your findings.

LabA5.java

26. Revert back to LabA1 and save it as LabA5. Modify the app so that after it parses the integer `x`, it computes and outputs the value of bit #10 of `x`. It does this using logical shifts and the following logic:

```
int z = x << 21;    // make bit #10 the MSb
z = z >>> 31;       // make bit #10 the LSb
```

    For example, if the supplied argument were 5000, then the output would be 0, and if it were 6000, then the output would be 1.

27. Add the following fragment to LabA5:

```
int mask = 1024;
int y = x & mask;
y = y >> 10;
```

28. The number, 1024, the mask, has the following representation in binary:

    0000 0000 0000 0000 0000 0100 0000 0000

    Verify that this mask-based approach will also output the state of bit #10; i.e. the mask-based `y` and the shift-based `z` are the same.

29. The word "mask" is often used to describe something that blocks the view except for a hole through which light can pass and one can see. Does the number 1024 behave like a mask in this sense?

LabA6.java

30. Revert back to LabA1 and save it as LabA6. Modify the app so that after it parses the integer `x`, it *sets* bit #10 of `x` (i.e. makes it 1) and *clears* bit #11 (i.e. makes it 0).

31. Have you program output `x` before and after the manipulation in binary so you can easily track its progress and debug it if needed. Report your results.

LabA7.java

32. Revert back to LabA1 and save it as LabA7. Modify the app so that after it parses the integer `x`, it interchanges (i.e. swaps) bit #10 and bit #20 of `x`.

33. Have you program output `x` before and after the manipulation in binary so you can easily track its progress and debug it if needed. Report your experiment.

LabA8.java

34. Revert back to LabA1 and save it as LabA8. Unlike all the earlier explorations, let us supply a negative value such as `-5` as argument The output should be:

    11111111111111111111111111111011

35. The representation of negative numbers is also positional, and in fact identical to that used for non-negative integers, except for one key difference: *the weight of the MSb is negative*. The above pattern, for example, represents:

    $$-2^{31} + 2^{30} + 2^{29} + 2^{28} + 2^{27} + \ldots + 2^{3} + 2^{1} + 2^{0} = -5$$

36. Have the app compute and output the value of:

    ```
    int z = 1 + ~x;
    ```

    both in decimal and binary. Run the program for various negative arguments and examine the output. What do you observe? Can you explain this observation?

LAB A
Notes

- Replicating the MSb of a pattern (aka *sign extension*) does not change the value of the integer it represents. This is obvious if the MSb = 0 but it is also true if it is 1.

- Even though the MSb is not a sign bit per se, it acts as one. If the MSb is 1, the value of the integer will work out to be negative.

- Arithmetic right shifts are intended to maintain the division interpretation of logical right shifts when the operand is negative.

- This lab is concerned with *signed* integers, ones that can have a sign. If an integer cannot have a sign (such as a memory address) then we call it *unsigned* and in that case the weights of all the bits, including the MSb, would be positive.

- Java's integer types (`byte`, `short`, `int`, and `long`) represent integers as signed two's complement numbers using 8, 16, 32, and 64 bits, respectively.

- Java does have an unsigned integer type, `char`, which uses 16 bits, but because of automatic promotion all integer operations end up being signed unless explicitly cast.

- When you approach problems involving bits, it is essential that you think it terms of shifts and masks, ***not*** numbers or strings. For example, a mask with bit #5 set should be constructed using `1 << 5` rather than `Math.pow(2, 5)`. Similarly, the state of a given bit should be determined based on logical operations rather than `substring` or `indexOf`. Relying on numbers or strings is not transferable to other languages and can in fact be wrong sometimes, when casting `Math.pow(2,31)`!