Bit-String Flicking Worksheet
Created by Sam Mercier and Sam Craig for the West Lafayette High School
ACSL Club of 2014–2015

There are four fundemental bit string operators that are incredibly common. They are **AND**, **OR**, **XOR**, and **NOT**. The first three are operators that take **two** operands. The last one, **NOT**, takes **ONE** operand. The operand can be a single bit, or it can be a bit-string. If it is a bit string you apply the operation to each bit going from **right to left**. You might be wondering, now, what exactly these operators do. We show this via something we call a truth table. It is just a set of example cases that show you what the output would be in every possible case. Here are their truth tables:

| p | q | p **AND** q | p **OR** q | p **XOR** q | **NOT** p |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

There is another set of operators. These are called the **shift operators** because they shift around a **bit-string**. There are two main groups of these operators. There are the **logical shift operators** which contains the operator **LSHIFT-n** and **RSHIFT-n**, and then there are the **circular shift operators** which contains the operators **LCIRC-n** and **RCIRC-n**. These operators have **two** operands. The first one is an **integer** which is attached to the operator by a hyphen (-). This indicates the number of times to do the operation. The second operand comes after the operator also and is separated by a space. This operand is the **bit-string** to be operated on.

It is probably not surprising that these operators shift the bits around. The question is how do they shift them around. Well the logical shift operators are a pure shift in that the bits that are moved outside of the string are lost, and extra empty spaces are filled with zeros. The CIRC shifts circulates the bits around. So bits that leave from the right side come back on the left side. This sounds much more complicated probably than it actually is so let me just demonstrate. In this table the **n** is always going to be 2 so that we can focus on how each operator effects the bit-string.

| p | **LSHIFT-2** p | **RSHIFT-2** p | **LCIRC-2** p | **RSHIFT-2** |
|---|---|---|---|---|
| 010101 | 010100 | 000101 | 010101 | 010101 |
| 111010 | 101000 | 001110 | 101011 | 101110 |
| 111111 | 111100 | 001111 | 111111 | 111111 |

**Questions**

1. Find the final bit-string after the following operations:
   `(NOT (LCIRC-3 (RSHIFT-2 (10011 XOR 10111))))`
2. Find the final bit-string after the following operations:
   `(10011 AND (RSHIFT-3 00111))`
3. Find the final bit-string after the following operations:
   `((LCIRC-4 (10110 OR 11010)) XOR 10111)`
4. Find all values of $x$ that solve the following equation:
   `((LCIRC-2 x) AND 11011) = 01100`
5. Find the final bit-string after the following operations:
   `(00010 OR 10111 AND 11100)`

**Answers**

1.

```
   (NOT (LCIRC-3 (RSHIFT-2 (10011 XOR 10111))))
→ (NOT (LCIRC-3 (RSHIFT-2 (00100))))
→ (NOT (LCIRC-3 (00001)))
→ (NOT (01000))
→ 10111
```

2.

```
   (10011 AND (RSHIFT-3 00111))
→ (10011 AND 00000)
→ 00000
```

3.

```
   ((LCIRC-4 (10110 OR 11010)) XOR 10111)
→ ((LCIRC-4 11110) XOR 10111)
→ (01111 XOR 10111)
→ 11000
```

4.

```
   ((LCIRC-2 x) AND 11011) = 01100
→ ((LCIRC-2 abcde) AND 11011) = 01100
→ (cdeab AND 11011) = 01100
```

`11011 AND 01100 = 01000`, so $x$ must be, where asterisk (*) is whichever, 00010.

5.

```
   (00010 OR 10111 AND 11100)
→ (00010 OR 10100)
→ 10110
```