Exercise 1 - Binary Operations

The C language has bit manipulation mechanisms. For example, consider two variables \(x\) and \(y\) of type integer and the operator \(\oplus\) (xor). We denote the \(i\)th bit of \(x\) and \(y\) by \(x_i\) and \(y_i\) respectively. The result of \(x \oplus y\) is the word \(z\) such that \(z_i = x_i \oplus y_i\). The C operators are \& (and), \| (or), \^ (xor) et \^ (not).

Do not confuse logical operators such as \&\&, ||, etc. with operators for handling binary words. Note that \(4\&2\) is 0, \(4\&\&2\) is 1.

Binary operations can be condensed. So \(x=x\|2\) can be written \(x|=2\), and \(x=x\^=y\) can be written \(x^=y\).

The language also provides the right shift operators \(\gg\) or the left shift \(\ll\).

1. What does the following code do :
   \[ n \& (n-1) \]

2. In the following snippet \(c\) and \(n\) are integers.
   \[ \text{for } (c = 0; n != 0; n &= (n-1)) c++; \]
   What value does \(c\) take according to the values of \(n\) ?

   We will study a method which efficiently counts the number of 1s in a word of length \(2^k\) (for a \(k \geq 0\)), that is, in \(O(k)\) number of operations, assuming \(2^k\) is the size of a register. Let \(l \leq k\) and \(n\) be a word of length \(2^k\). We denote by \(l\text{-block}\) a block of \(2^l\) consecutive bits in \(n\), such that these blocks do not overlap. (For example, there are eight 2-blocks of length 4 in a 32-bit word.) The \(l\)-count of \(n\) is the word of length \(2^k\) such that each of its \(l\)-blocks contains the number of 1’s of the corresponding \(l\)-block in \(n\). Trivially, any word equals its own 0-count. We try to produce the \(k\)-count of \(n\). In what follows, we will assume that \(k = 5\), and suddenly we are working with 32-bit registers. The method is, however, easy to generalize.

3. Find an operation that produces the 1-count of \(n\) (in constant time).

4. Generalize and iterate this operation to calculate the 5-count of \(n\).

   We work with 64-bit registers. Let \(n = (stuvwxyz)\) be a byte, with \(s\) the most significant bit and \(z\) the least significant.

5. What does the following C expression give?
   \[ (n * 0x0202020202 & 0x010884422010) \% 1023 \]

Exercise 2 - De Bruijn sequences

In this part of the TD, we will develop an efficient method to count the number of trailing zero bits in a given (unsigned) integer value \(x\) such that \(x > 0\). Equivalently, we can compute the position of the least significant bit whose value is 1. For example, if the binary representation of \(x\) is 10110100, then the bit we are looking for is the 1 which is followed by the two final 0s.

An index in a bit string is identified from right to left starting at zero. E.g., for \(x = (10110100)\), the bits of \(x\) at index 0 and 1 are 0, and the bit with index 2 is 1. We present this method for \(2^3 = 8\) bit words, but it can be generalized to \(2^n\) bits for any \(n > 0\).
Given $x \in \mathbb{N}$ such that $0 < x < 2^8$, we will be interested in implementing a function $\ell : \{1, \ldots, 2^8 - 1\} \to \{0, \ldots, 7\}$ such that $\ell(x)$ is equal to smallest index that is set to 1 in the binary representation of $x$. In the example above, we have $\ell(x) = 2$.

1. Write a naive C function to solve this problem (skeleton below).

```c
unsigned int l (unsigned int x) { // we assume 0 < x < 256
    int result = 0;
    ...
    return result;
}
```

However, the running time of this function depends on the number of bits in $x$. We will develop another algorithm has constant running time, i.e. independent of the actual number of zeros. To this end, we study de Bruijn sequences.

A de Bruijn sequence $s(n)$ of order $n$ is a cyclic bit string such that every binary string of length $n$ occurs exactly once in $s$. Cyclic means that once you reach the end of $s(n)$ you may continue at the beginning of $s(n)$. For example, for $n = 2$ we can set $s(n) = 0011$ since 00, 01, 10 and 11 can all be found in $s(n)$; in particular 10 starts at index 0 of $s(n)$ and then continues at index 3 of $s(n)$.

2. Give a lower bound for the minimal length of a de Bruijn sequence $s(n)$.

De Bruijn sequences can be obtained from paths in de Bruijn graphs. The vertices of a de Bruijn graph of order $n$ are all bit strings of length $n$. There is a directed edge between two vertices $b_1b_2\cdots b_n$ and $c_1c_2\cdots c_n$ if and only if $b_2 = c_1$, $b_3 = c_2$, $\ldots$, $b_n = c_{n-1}$.

The figure 1 depicts the de Bruijn graph of order 2.

![Figure 1 - De Bruijn graph of order 2.](image)

3. Draw the de Bruijn graph of order 3.

A de Bruijn sequence can be obtained from a de Bruijn graph by following a Hamiltonian cycle that starts and ends in the vertex $0\cdots 0$. A Hamiltonian cycle is a cycle that visits each vertex exactly once before returning to the starting vertex. For instance, the only Hamiltonian cycle in the graph in the figure above is $00 \to 01 \to 11 \to 10 \to 00$. This cycle corresponds to the aforementioned de Bruijn sequence $0011$. One can in fact prove that such a Hamiltonian cycle exists in every de Bruijn graph.

4. Find two different de Bruijn sequences of order 3 by following two different Hamiltonian paths in your de Bruijn graph of order 3 starting in vertex 000.

5. Choose a de Bruijn sequence $s(3)$ of order 3 from the previous question and complete the following table:
6. Let $s(3)$ be the de Bruijn sequence from the previous question and $0 \leq j < 8$. What is the value assigned by the table of the bit string:

$$((s(3) \ll j) \gg 7) \& 0x7$$

Here, $\ll$ and $\gg$ mean shift-left and shift-right, respectively, and $\&$ is binary AND.

7. Given an unsigned integer $k > 0$, what is the value of $k \& (-k)$, where $-k$ is the two’s complement of $k$?

8. Propose an implementation of $\ell(x)$.

**Exercise 3 - Some logical components**

Recall the NAND gate: It is a logic gate which produces an output which is false only if all its inputs are true. We have its truth table below:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \uparrow q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The goal of this exercise is to implement other components, in an incremental fashion. This is the only component you can use at the start. Once you have implemented a component correctly, it will be usable for the implementation of future components. Try to optimize both, the least number of pre-defined components used, as well as the number of NAND-gates used.

1. **NOT**
2. **AND**
3. **OR**
4. **XOR**
5. Equal to Zero (input is a 4-bit word)
6. Bonus: You can assume you have the 16 bit components for the above functions, along with a 16-bit adder. Construct a SUBTRACTOR that subtracts $B$ from $A$ ($A-B$), where $A$ and $B$ are 16-bit numbers.

Truth tables for the above functions:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\neg p$</th>
<th>$p \land q$</th>
<th>$p \lor q$</th>
<th>$p \oplus q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>