TD 1 : Integers and Bit Representation

Solutions

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 ith 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) and `~` (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 $>>$ or the left shift $<<$.

1. What does the following code do:

```
n & (n-1)
```

Solution: Remove the least significant 1 (if $n \neq 0$). Equal to 0 if $n = 0$.

2. In the following snippet c and n are integers.

```
for (c = 0; n != 0; n &= (n-1)) c++;
```

What value does c take according to the values of n?

Solution: Example with $n = 5$.
$c = 0$; $n \neq 0$ so `n&(n-1)` or `5&4`, in binary `101&100`, so $n = (100)_2$ and $c = 1$.
n is greater than 0 so `4&3` or `100&011` so $n = 0$ and $c = 2$.
c is the number of 1s in the binary representation 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 $\mathcal{O}(k)$ 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 $l - block$ 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).

Solution: In one copy of `n`, we keep the bits in even positions, in another the bits in odd positions. This is done with the help of *masking*. We know that for any bit b, $0 \& b = 0$ and $1 \& b = b$. Hence, for the first copy, we `AND` the even positions with 1 (guarding) and the odd positions with 0 (masking). Similarly, for the second copy, we guard the odd positions and mask the even positions. By shifting one of the copies, we can add all the blocks at the same time.

```
((n & 0xaaaaaaaa) >> 1) + (n & 0x55555555)
```

Moreover, the additions are independent because any block is large enough to store the number of 1's of the original block. Hence, it is in constant time.

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

Solution: We progressively generate the 1-count, 2-count, etc.

```
n = ((n & 0xaaaaaaaa) >> 1) + (n & 0x55555555);
n = ((n & 0xcccccccc) >> 2) + (n & 0x33333333);
n = ((n & 0xf0f0f0f0) >> 4) + (n & 0x0f0f0f0f);
n = ((n & 0xff00ff00) >> 8) + (n & 0x00ff00ff);
n = ((n & 0xffff0000) >> 16) + (n & 0x0000ffff);
```

We work with 64-bit registers. Let $n = (stuvwxyz)_2$ be a byte, with s the most significant bit and z the least significant.

5. What does the following C expression give? How? (see program *bits.c*)

```
(n * 0x0202020202 & 0x010884422010) % 1023
```

Solution: This expression inverts the order of the bits. i.e. it gives $(zyxwvuts)_2$. See explanation here : `https://graphics.stanford.edu/~seander/bithacks.html#ReverseByteWith64BitsDiv`

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 $x = (10110100)_2$, 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).

```
int main (){
unsigned int x; // we assume 0 < x < 256
int result = 0;
```

```
... //to be filled

return result;

}
```

Solution:

```
int main (){
unsigned int x; // we assume 0 < x < 256
int result = 0;

while ((x & 1) == 0) {
result++;
x = x >> 1;
}

return result;

}
```

Remarque: Attention à bien parenthéser dans le while, le $\&$ n'a pas la priorité sur le $==$.

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. For example, for $n = 2$ we can set $s(n) = 00110$ since $00, 01, 10$ and 11 can all be found in $s(n)$.

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

Solution: There are 2^n different words in $\mathcal{B}^n$. A trivial lower bound for a sequence s is therefore of $2^n + (n - 1)$, since each word in $\mathcal{B}^n$ must start at a position other than s and must be followed by $n - 1$ more bits.

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 $b_1 b_2 \cdots b_n$ and $c_1 c_2 \cdots c_n$ 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.

3. Draw the de Bruijn graph of order 3.

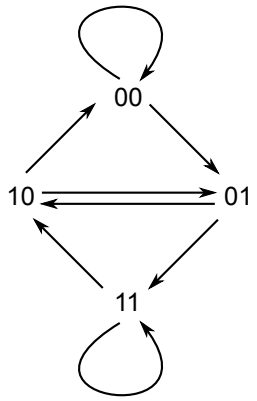
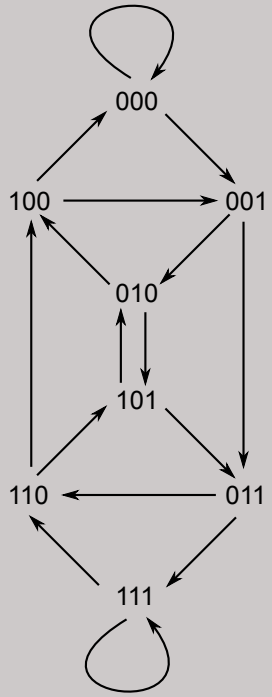


Figure 1: De Bruijn graph of order 2.

Solution:



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 00110. 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.

Solution:

$000 \to 001 \to 010 \to 101 \to 011 \to 111 \to 110 \to 100$

0001011100

$000 \to 001 \to 011 \to 111 \to 110 \to 101 \to 010 \to 100$

0001110100

5. Choose a de Bruijn sequence $s(3)$ of order 3 from the previous question and complete the following table:

bit-string	7- index in $s(3)$
000	0
001	
010	
011	
100	
101	
110	
111	

Solution:

Using the sequence 0001011100, we obtain the table below We observe that the right column is a permutation of $\{0, \ldots, 7\}$.

bit-string	index in s
000	0
001	1
010	2
011	4
100	7
101	3
110	6
111	5

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.

Solution: Let $e(j) = ((s(3) \ll j) \gg 7)$ & `0x7` The relation between $e(j)$ and j is exactly the table given above, which is bijective. Hence, for $e(j) = 000$ we have $j = 0$, etc.

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

Solution: Refresher: The two's complement is obtained by flipping (inverting) the bits in binary and adding 1.

Example : 4 : 00000100 ; flipping-> 11111011 ; 2's complement(+1) : 11111100 Hence, if the binary representation of x is $c_0 c_1 \cdots c_i 10 \cdots 0$, the negation is $\bar{c_0}\bar{c_1} \cdots \bar{c_i} 01 \cdots 1$, and the representation of $-x$ becomes $\bar{c_0}\bar{c_1} \cdots \bar{c_i} 10 \cdots 0$, and the expression then has only one non-zero bit, at position $\ell(x)$, which evaluates to $2^{\ell}(x)$.

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

Solution: We use the De Bruijn sequence to build up a multiply-right-shift perfect hashing algorithm to index the LSB of the given integer, and use that index to return LSB index of the given integer. The first step is to isolate the LSB. As we saw from the previous question, in order to do that we compute $k \ \& \ (-k)$. Next we multiply this by the De Bruijn sequence. A good De Bruijn sequence must hash all possible power-of-two integers (i.e. that have only one bit set) uniquely.

Multiplying by a power of 2 is equivalent to a shift. If the input to the hash function has a bit on in position i, then the multiplication causes `debruijn` to be shifted left by i positions. . Each of the n possible shifts causes the top $log_2 n$ bits of the resulting n-bit word to take on a distinct value.

Next, we shift by the required bits ($n - log_2 n = 8 - 3 = 5$). Shifting these $log_2 n$ bits into the low-order bits of the word allows us to index the table mapping the “de Bruijn index" into the normal index.

See `debruijn.c` on the Teaching page for the complete implementation in C.

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:

p	q	$p \uparrow q$
0	0	1
0	1	1
1	0	1
1	1	0

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.