Partiel d'Architecture et Système

14 novembre 2014

Duration: 2 hours. Answers can be given in either English or French. Justify all your answers. The computers in the room cannot be used during the exam.

There is a maximum of 30 points to be gained in the questions.

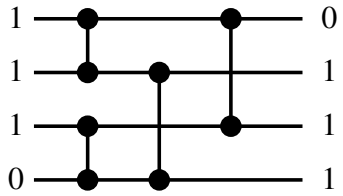
The final mark for the course will be either the average of the marks for this exam and the final exam, or the mark of the final exam, whichever is better.

1 Sorting networks

We first recall some basic facts about *sorting networks*, as treated in one of the exercises.

The purpose of a sorting network is to sort n numbers in ascending order. For a given network, n is fixed. A network is called *correct* if for all possible inputs, the output is sorted, with the lowest element at the top and the highest at the bottom. It is known that a sorting network is correct iff it is correct for all sequences consisting of 0 and 1 (zero-one principle), so we shall assume that all inputs are 0 or 1.

The basic element of a sorting network is a *comparator*. It takes two values and yields two outputs, the upper line being the smaller and the lower line the bigger of the two values. A sorting network consists of n wires, for a fixed n, connected by comparators. For convenience, we shall draw connectors simply as vertical wires. Below is a sorting network for $n = 4$ and an example of its input and output.



Prove or refute the following statements. It is not enough to say "yes" or "no", you need to justify your response. Every answer is worth 2 points.

(i) In every correct network there is at least one comparator between every pair of neighbouring wires.

(ii) Every network that contains at least one comparator between every pair of wires is correct.

(iii) A correct network stays correct if one adds any comparator at the end.

(iv) A correct network stays correct if one adds any comparator anywhere in the network.

2 Floating-point numbers

We recall some basic facts about floating-point representations. A floating-point number consists of three components, the *sign*, the *exponent*, and the *mantissa*. The value of a triple (s, m, e) is thus $\pm 2^e \cdot m$.

The sign is represented by a single bit, which is 0 for positive and 1 for negative. We speak of a (k, ℓ)-format if k bits are used for the exponent and ℓ bits for the mantissa. Thus, the format used for the `float` data type in C would be a $(8, 23)$-format. As in the IEEE-754 standard we assume that if the bit pattern used for the exponent, interpreted as an unsigned integer, has value E, then $2^{k-1} - 1$ must be subtracted from E to obtain the exponent value e. Thus, in the `float` data type, a value of 128 actually means an exponent of 1. As for the mantissa, its value is in the range $[1, 2)$, where the most significant bit has a weight of $1/2$, the second most significant bit $1/4$, etc. Thus, if $\ell = 3$ and the bit pattern of the mantissa is 101, then the represented value is $1 + 1/2 + 1/8$.

Note: The representations of 0, infinity, and "not a number" will be irrelevant for the following exercises.

Due to the limited number of bits that are available for mantissa and exponent, not all real values (not even all rational values) can be represented exactly.

(a) (3 points) Assume a $(4, 7)$-format for floating-point numbers, and provide the bit patterns for the values 2.5, -42, and 12.34. If any of these numbers is not exactly representable in this format, round it to the nearest value that is.

(b) (2 points) In the $(4, 7)$-format, what is the smallest positive integer that cannot be represented exactly? What is it in the $(3, 7)$-format?

(c) (2 points) Given some (k, ℓ)-format, let $N(k, \ell)$ denote the smallest positive integer that cannot be represented exactly in this format. For a fixed ℓ, give a formula for the minimal value of k such that for any $k' > k$, $N(k, \ell) = N(k', \ell)$.

3 De Bruijn sequences

In this part of the exam, 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. Incidentally, one concrete application – when x has 64 bits – is to encode the positions of pieces on a chess board and iterate over these. Here however, we will simplify matters by assuming that x has only 8 bits.

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.

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 $\ell(x) = 2$.

In principle, we could solve the problem using the C function below, which shifts x to the right until the least significant bit is 1.

```
unsigned int l (unsigned int x) {   // we assume 0 < x < 256
  int result = 0;
  while (x & 1 == 0) {
    result++;
    x = x >> 1;
  }
  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 will 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 $s(n)$.

An obvious lower bound for the minimal length of a de Bruijn sequence $s(n)$ is 2^n. We will see that a sequence of this length can always be found, then use it to achieve our initial goal.

(a) (3 points) 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 below depicts the de Bruijn graph of order 2.

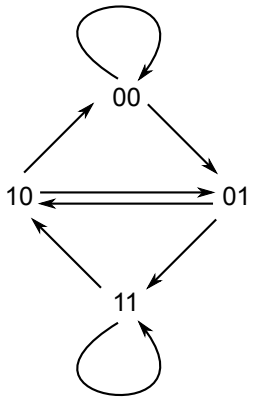


Figure 1: De Bruijn graph of order 2.

Draw the de Bruijn graph of order 3.

(b) (2 points) 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.

Read off 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.

(c) (2 points) *Choose a de Bruijn sequence $s(3)$ of order 3 from (b) and complete the following table:*

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

(d) (2 points) *Let $s(3)$ be the de Bruijn sequence from (c) and $0 \leq j < 8$. What is the value assigned by the table in (c) of the bit string*

$$((s(3) \ll j) \gg 5) \;\&\; 0\text{x}7$$

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

(e) (2 points) *Given an unsigned integer $k > 0$, what is the value of k & $(-k)$, where $-k$ is the two's complement of k?*

(f) (3 points) *Complete the following code skeleton such that it computes $\ell(x)$:*

```
const int index[8] = { 0, ... }; // the right-hand side of the table
                                 // in (c) here
const int s3 = 0b...;            // your de Bruijn sequence used in (c)

unsigned int l(unsigned int x) { // we assume 0 < x < 256
  return index[ ... ];           // complete code in the brackets
}
```

(g) (1 point) What other advantage does the algorithm above have with respect to our initial while loop, besides being constant runtime? (Think of the way assembly code is executed in a CPU, and how modern processors try to optimize that execution.)