Input File: `register.in` **100 Points**
Output File: `register.out` **Time limit:** 1 s
Source File: `register.pas/.c/.cpp` **Memory limit:** 16 MB

Idea

Express the output of the XOR gate as a linear function in $\mathbb{Z}_2 = \{0, 1\}$. This results in a system of n linear equations. Solve them using a simplified version of Gauss' Algorithm.

Solution

First of all, we notice that when calculating in $\mathbb{Z}_2$ (i.e. modulo 2), multiplication is the same as AND, and addition is the same as XOR. This allows us to describe the combined effect of the switches S_i and the XOR gate using the linear expression

$$s_1\, a_1 + s_2\, a_2 + \ldots + s_n\, a_n.$$

Let o_i ($i \geq 0$) denote the i-th output bit (which is also the i-th value on line 2 of the input file). The first n output bits $o_0 \ldots o_{n-1}$ are simply the initial register bits a_i. The first bit with any calculation involved is o_n. When o_n is calculated, the register bits a_i are occupied (in reverse order) by the values $o_0 \ldots o_{n-1}$. Using the linear expression we constructed, this yields our first equation:

$$o_n = s_{n-1}\, o_0 + s_{n-2}\, o_1 + \ldots + s_0\, o_{n-1}.$$

Similar equations hold for $o_{n+1} \ldots o_{2n-1}$. To make the equations even easier to handle, lets reverse the numbering of the switches by defining $s'_i := s_{n-1-i}$. Our equations now look like this:

$$\begin{array}{llll} s'_0\, o_0 & + s'_1\, o_1 & + \ldots + s'_{n-1}\, o_{n-1} & = o_n \\ s'_0\, o_1 & + s'_1\, o_2 & + \ldots + s'_{n-1}\, o_n & = o_{n+1} \\ & \vdots & \vdots & \vdots \\ s'_0\, o_{n-1} & + s'_1\, o_n & + \ldots + s'_{n-1}\, o_{2n-2} & = o_{2n-1} \end{array}$$

When the equations are written as a matrix, we do not need the s'_i explicitly. The resulting matrix $M = (m_{i,j})$ has n rows for n equations and $n+1$ colums for n variables and the right-hand side. From our equations above we see that $m_{i,j} = o_{i+j}$.

Forward Elimination Start with $r := 0$. For each column i, make sure there is at most one non-zero element in rows $\geq r$. If there is none, the column is ignored—the system is under-determined and we will be able to choose an arbitrary value for the variable s'_i.

If there is at least one "1", swap the relevant rows to move it to row r, and cancel out any additional "1"s in rows $> r$ by adding row r to that row. Row r is the *defining equation* for s'_i. Increment r.

Existence of Solutions The only case where no solution can be found is the case where all coefficients of an equation are 0, but the right-hand side is 1. This condition is checked for each row of M. If there is no solution, "`-1`" is written to the output file.

Backward Substitution Loop backwards over the variables s'_i. If no defining equation for s'_i was found during forward elimination, choose 0 as the value for that variable.

If a defining equation r exists, then row r of M looks like this:

$$\underbrace{0\ldots0}_{i}\ 1\ *\ *\ *\ **\ =\ *.$$

All coefficients left of column i are 0—this is guaranteed by the forward elimination process. Element i is 1, and elements $> i$ can have any value. But since we are looping backwards, we have already assigned values for all variables $> i$, so we can simply calculate s'_i by subtracting (modulo 2 that is the same as adding) all values of variables whose coefficients are 1 from the right-hand side value. Because the s_i that are to be written to the output file are just the s'_i in reverse, they can be printed directly in the backward substitution loop.

Complexity and Runtime

Forward elimination is in $O(N^3)$, backward substitution is in $O(N^2)$, so the overall complexity of the algorithm is in $O(N^3)$. The slightly simpler full elimination could be used instead of forward elimination and backward substitution, however this roughly doubles the amount of row additions required. Therefore, using backward substitution leads to a program that is almost twice as fast as an otherwise identical program using full elimination.