

Lab 8: Bitwise operations - Working on K bit integers

CS1234: Small Scale Application Development

=====

C Programming: Problem statement: (5 marks)

You are required to store multiple K bit numbers in a single 64 bit uint64_t C integer. This single integer will act as your "array" for storing the K bit integers. Initially, it is set to 0.

K will be a power of 2, and K will lie between 1 and 64, both inclusive. The numbers are supposed to be treated as unsigned and positive.

For eg:

To store a 4 bit number in our 64 bit "array", we treat the 64 bits as an array of 4 bit numbers. The least significant 4 bits, i.e., the 4 bits on the right of the number, form the 0th element of the array.

Let us say that the 64bit contents were:

1111 1100 ... 1001

Now we store 1101 at index 0. The contents become:

1111 1100 ... 1101

You are supposed to implement the following operations:

- insert <index> <number>

This inserts the specified 'number' at the specified 'index'.

The 'index' and the 'number' will always be valid.

- search <number>

This searches for the 'number' in the array, printing the index if found, and -1 otherwise. If multiple entries are present, print the smallest index found.

- printAll

Print all the K bit integers stored, each one on a new line, starting from the index 0 and going up to the maximum index.

- lShift <index> <shift>

Left shift the integer at the given 'index' by 'shift' amount. If the integer overflows, round it by the rules of unsigned integer rounding.

Example: let K be 4, and the number be 1010. If we do an lshift by 2 units, the number (assuming more than 4 bits) becomes 101000. However, since we only have 4 bits per storage, the number is bit-truncated to 1000. It might be relevant to know that this is a precise specification, while C treats overflows as undefined behaviour, and may do anything.

- rShift <index> <shift>

Right shift the integer at the given 'index' by 'shift' amount. Underflows over here are handled similar to how overflows are handled in lshift. The bits that 'fall off' are just disregarded. For instance, if K is 4, and the number is 1001, and we're right shifting by 2, then the number becomes 0010.

- add <index> <number>

Add the given 'number' to the integer stored at 'index', and store the K bit sum in 'index' again. 'index' and 'number' will be valid. Again, overflow is treated the same as that in lshift.

Input format:

The input will consist of multiple lines read through STDIN. The first line will contain K, the width of the integers to be stored. Each subsequent line will be of one of the following formats:

- insert <index> <number>
- search <number>
- printAll
- lShift <index> <shift>
- rShift <index> <shift>
- add <index> <number>

These are the operations as described above

Output format:

Only printAll and search prints the output to STDOUT as described above.

An example:

Suppose the contents were all 16 bit long, and were as follows:

1111000011110000 1111000000001111 1111111111111111 0000111100001111

The output for printAll would be:

3855

65535
61455
61680

You can check that they correspond to the 16 bit numbers stored from the lowest (rightmost) index, to the highest (leftmost) index.

Constraints:

- The width K of the integers will be in the range 1 to 64, both inclusive
- The width K will be a power of 2
- Any index given in a command will lie between 0 and (64 / K), 0 inclusive.
- Any number given in a command will be given as its binary representation, including the leading zeros. That is, the length of the number will be exactly K bits. Note that this is only for the numbers in insert, search and add functions.

Sample input/output:

INPUT:

4
insert 0 1111
insert 1 1100
add 1 0100
insert 2 1101
lShift 2 2
insert 3 1011
rShift 3 2
printAll

OUTPUT:

15
0
4
2
0
0
0
0
0
0
0
0
0
0
0
0

Notes and Submission Guidelines:

- Local testing: Run `check.sh` to check the testcases locally.
- Submission:
 - Change your roll number in `configure.sh`.
 - Run `check.sh` to check your code against the public testcases.
 - Run `submit.sh` to submit the lab.

Extras (ungraded):

Extension 1: multiplication of K bit integers

Similar to `add`, we now define `mul <index> <othernum>` to multiply two K bit integers, and store the result in the same index. Overflows are treated as they were in the rest of the commands.

Extension 2: Karatsuba multiplication

In due course of time, in the algorithms course, you will come to learn that the naive multiplication we do on pen and paper is relatively inefficient.

Implement Karatsuba multiplication, which is a more efficient form of multiplication.

Let the numbers a, b be of K bits, where K is a power of 2.

We split both the numbers into 2 parts, a_1 and a_0 , where a_1 has the upper $K/2$ bits of the number a , and a_0 has the lower $K/2$ bits of the number a . Similarly, we define b_1 and b_0 .

let K be 2^p

Now, the product $(a * b)$ can be computed as follows:

$$\begin{aligned}(a * b) &= ((a_1 \ll (K / 2)) + a_0) * ((b_1 \ll (K / 2)) + b_0) \\ &= ((a_1 * b_1) \ll K) + ((a_0 * b_1 + b_0 * a_1) \ll (K / 2)) + (a_0 * b_0) \\ &= (a_1 * b_1) \ll K \\ &\quad + ((a_0 + a_1) * (b_0 + b_1) - (a_1 * b_1) - (a_0 * b_0)) \ll (K / 2) \\ &\quad + (a_0 * b_0)\end{aligned}$$

If you observe carefully, you will notice that it actually decreased the number of multiplications needed to compute the final result.

Given two K bit integers, implement karatsuba multiplication for them as specified above, and then use it for the previous extension.