Data Lab: Manipulating Bits

CS 105, Fall 2018

Due on Tuesday, September 17, 2019, at 11:59 PM

The purpose of this assignment is to give you familiarity with bit-level representations of integers and floating point numbers. You will accomplish the goal by solving a series of programming "puzzles." Even though many of the puzzles are quite artificial, you will find yourself thinking much more about bits in working your way through them.

Logistics

You must work in a group of two people in solving the problems for this assignment. Any clarifications or revisions will be posted on Piazza. *We strongly recommend that you and your partner brainstorm before coding!*

Getting Started

The materials for the data lab are already on the server. First `ssh` into `pom-itb-cs2` and connect its files to your local computer with `sshfs`. Then create a (protected) directory in which you plan to do your work and change to it. Next give the command[1]

```
% tar xvf /data/datalab-handout.tar
```

which will cause a number of files to be unpacked in the directory. The only file you will be modifying and submitting is `bits.c`.

Begin by opening the file in an editor and put your names and userids in the comments at the top of the `bits.c` file. Do this right away!!

The `bits.c` file contains a skeleton for each of the 15 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code (no loops or conditionals) and a limited number of C arithmetic and logical operators. Each function heading tells you what operations are allowed. Further, you are not allowed to use any constants longer than 8 bits. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

[1] A note on color-coding: In a few cases we display commands that you type in a terminal window and the resulting output. We use % for the prompt. The characters that you are to type are in green, and the system's responses are in purple.

Compiling the Code

Do your work on `pom-itb-cs2`. You can be sure that the support programs `btest` and `dlc` will work there. In any case, make sure that the version you turn in compiles and runs correctly on `pom-itb-cs2`. If it fails to compile there, we cannot grade it.

Check the file `README` for documentation on running the `btest` program. You will find it helpful to work through the functions one at a time, testing each one as you go. You can use the `-f` flag to instruct `btest` to test only a single function, as in `./btest -f isPositive`.

Dig more deeply into the `README` file for information on some helper programs.

We have given you a `Makefile` to ease the burden of running the compiler. Type

```
% make btest
```

to compile the program `btest`, or simply

```
% make
```

to compile everything.

The `dlc` Program

The `dlc` program, a modified version of an ANSI C compiler, will be used to check your programs for compliance with the coding style rules. The typical usage is

```
% ./dlc bits.c.
```

- Type `./dlc -help` for a list of command line options. The `README` file is also helpful.
- The `dlc` program runs silently unless it detects a problem.
- Do not include `<stdio.h>` in your `bits.c` file, as it confuses `dlc` and can result in some non-intuitive error messages.
- Running with the `-e` switch causes `dlc` to print counts of the number of operators used by each function.
- ANSI C, and hence `dlc`, disallows `//` comments.
- In ANSI C, you must make all variable declarations at the beginning of a function. The following code is not accepted by `dlc`.

```
int mask = 0x55 + (0x55 << 8);
mask = mask + (mask << 16);
int shift = (x >> 1);
int sum = (shift & mask) + (x & mask);
```

- You may ignore the warning about a "non-includable file."

Evaluation

Your code will be run and tested on `pom-itb-cs2`. Your score will be computed out of a maximum of 72 points. We will use the programs `driver.pl` and `dlc`, supplied with the laboratory materials, to evaluate your code.

- Each function will be evaluated separately for correctness and performance. No points will be given for a function if `dlc` reports an illegal operator, too many operators, or another error. Otherwise, the function will be given the correctness and performance points assigned by `driver.pl`.
- Your `bits.c` file will be evaluated by the graders and given up to 5 points for style. For this laboratory, "good style" is easy to attain. It means that your names are present at the top of the file, that your code is understandable and consistently indented, that comments—when necessary to explain—are present and easy to read, and that there is no extraneous material.

Submission Instructions

When you have finished, submit one file, `bits.c`, to the course submission page.

- Make sure you have included all your team members' names in your file `bits.c`.
- Remove any extraneous print statements before submitting the file.
- Use the submission system, linked from the course web page, to submit the file `bits.c`. Use all the team members' names when submitting.
- If you discover a mistake in your code, simply submit the file again.

Part I: Bit Manipulations

Table 1 describes a set of functions that manipulate and test sets of bits. The "Rating" field gives the difficulty rating (the number of points) for the puzzle, and the "Max Ops" field gives the maximum number of operators you are allowed to use to implement each function.

Function `bitNor` computes the bitwise NOR function. That is, when applied to arguments `x` and `y`, it returns `~(x|y)`. You may only use the operators `&` and `~`. Similarly, function `bitXor` should duplicate the behavior of the operation `^`, using only the operations `&` and `~`.

Function `isNotEqual` compares `x` to `y` for inequality. As with all *predicate* operations, it should return `1` if the tested condition holds and `0` otherwise.

Function `getByte` extracts a byte from a word. The bytes within a word are ordered from 0 (least significant)

Name	Description	Rating	Max Ops
`bitNor(x,y)`	`~(x|y)` using only `&` and `~`	1	8
`bitXor(x,y)`	`^` using only `&` and `~`	1	14
`isNotEqual(x,y)`	`x != y`?	2	6
`getByte(x,n)`	Extract byte `n` from `x`	2	6
`copyLSB(x)`	Set all bits to LSB of `x`	2	5
`logicalShift(x,n)`	Logical right shift `x` by `n`	3	16
`bitCount(x)`	Count number of 1's in `x`	4	40
`bang(x)`	Compute `!x` without using `!` operator	4	12
`leastBitPos(x)`	Mark least significant 1 bit	2	6

Table 1: Bit-Level Manipulation Functions.

to 3 (most significant). Function `copyLSB` replicates a copy of the least significant bit in all 32 bits of the result. Function `logicalShift` performs logical right shifts. You may assume the shift amount n satisfies $0 \leq n \leq 31$.

Function `bitCount` returns a count of the number of 1's in the argument. Function `bang` computes logical negation without using the `!` operator. Function `leastBitPos` generates a mask consisting of a single bit marking the position of the least significant one bit in the argument. If the argument equals 0, it returns 0.

Part II: Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers.

Function `tmax` returns the largest integer.

Function `isNonNegative` determines whether `x` is less than or equal to 0.

Function `isGreater` determines whether `x` is greater than `y`.

Function `divpwr2` divides its first argument by 2^n, where n is the second argument. You may assume that $0 \leq n \leq 30$. The function must round toward zero.

Function `absVal` is equivalent to the expression `x<0?-x:x`, giving the absolute value of `x` for all values other than *TMin*.

Function `addOK` determines whether its two arguments can be added together without overflow.

Name	Description	Rating	Max Ops
`tmax(void)`	largest two's complement integer	1	4
`isNonNegative(x)`	`x >= 0?`	3	6
`isGreater(x,y)`	`x > y?`	3	24
`divpwr2(x,n)`	`x/(1<<n)`	2	15
`absVal(x)`	absolute value	4	10
`addOK(x,y)`	Does `x+y` overflow?	3	20

Table 2: Arithmetic Functions