CSE 351, Autumn 2010
Lab 1: Manipulating Bits
Due: Thursday, October 7, 11:59PM

Questions? Problems? See the course website for contact info:
`cs.washington.edu/351`

Introduction

The purpose of this assignment is to become more familiar with bit-level representations and manipulations. You'll do this by solving a series of programming "puzzles." Many of these puzzles are quite artificial, but you'll find yourself thinking much more about bits in working your way through them.

Logistics

For this assignment, please work alone. Any clarifications and revisions to the assignment will be posted on the course web page. We will assume that you are working on department Linux machines. We suggest any of the Linux workstations in the Allen Center 002, 006, and 022 labs, as well as `attu`, a machine you can access remotely via SSH. *Links to info on using these systems can be found on the course web page.* The assignments and the tools that you will be using to complete them have all been tested in this environment. Submission and grading of your work will also happen here.

You may also choose to work remotely using a virtual machine. The CSE lab prepares virtual machine images that, in a rough sense, allow you to take a lab Windows or Linux workstation home with you – you simply boot the virtual machine as an application on your personal system(s). Instructions for downloading the appropriate software and image are available on the course website.

Tips for Working on Your Own Machine You are welcome to do your code development using any system or compiler you choose, but make sure the final product works on one of the department Linux machines. We can't guarantee that everything will work on your own computer, but we can point you in the right direction. For this lab we'll be using the GNU C Compiler (`gcc`) version 4.4.1 and GNU `make` version 3.81, but any recent versions of these tools should suffice. The `dlc` tool provided with the lab handout is compiled for the department Linux machines, though it may work elsewhere too (most likely on 32-bit x86 Linux machines).

Instructions

The files you will need for this assignment are packaged as `lab1-handout.zip`, available on the course website or on department Linux machines at:
`/projects/instr/10sp/cse351/labs/lab1-handout.zip`

Start by copying `lab1-handout.zip` to where you plan to do your work, then extract it. This can be done on the Linux command line as follows:

```
[you@attu3 ~]$ mkdir 351
[you@attu3 351]$ cd 351
[you@attu3 351]$ cp /projects/instr/10sp/cse351/labs/lab1-handout.zip .
[you@attu3 351]$ unzip lab1-handout.zip
   ... some output ...
```

If you are working on a CSE Windows machine, you should save and unzip `lab1-handout.zip` somewhere in your Z: drive. We stronly recommend using the CSE Lab virtual machine software and images if you want to run the assingment from a Windows machine. Instructions for setting this up are available on the course website.

Once you've unzipped `lab1-handout.zip`, you'll have a directory called `lab1-handout` containing several files. The only file you will be modifying and submitting is `bits.c`. (Feel free to look at the other files, but any modifications you make outside `bits.c` will not be seen when grading.)

The file `btest.c` allows you to evaluate the functional correctness of your code. The file `README` contains additional documentation about `btest`. Move into the lab directory and use the command `make` to generate the test code and run it with the command `./btest`:

```
[you@attu3 351]$ cd lab1-handout
[you@attu3 lab1-handout]$ make
   ... some output ...
[you@attu3 lab1-handout]$ ./btest
   ... some output ...
```

Before you have done any work, `btest` will spit out lots of information about tests that fail. The file `dlc` is a tool that you can run to check your solutions for compliance with the coding rules. The remaining files are used to build `btest`.

The `bits.c` file contains a skeleton for each of the 15 programming puzzles (2 of which are extra credit). Your assignment is to complete each function skeleton using only *straight-line* code (i.e., no loops or conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are *only* allowed to use the following eight operators:

```
! ~ & ^ | + << >>
```

A few of the functions further restrict this list. Also, 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. The tasks are described in more detail below.

Every time you want to test your functions with `btest`), first run `make` to recompile. This takes care of running the GCC compiler with the right options, so if there are any compilation errors with your functions in `bits.c`, they will show up when you run `make`.

Evaluation

Your code will be compiled with GCC and run and tested on one of the class machines. Your score will be computed out of a maximum of 65 points based on the following distribution:

32 Correctness of code running on one of the class machines.

26 Performance of code, based on number of operators used in each function. (2 points each)

7 Style points, based on your instructor's subjective evaluation of the quality of your solutions and your comments.

Extra credit (up to 8 points) will be awarded as follows:

4 Correct implementation of `bitCount` and `leastBitPos`. (2 points each)

4 Performance of `bitCount` and `leastBitPos`, based on number of operators used in each function. (2 points each)

The puzzles have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 32. We will evaluate your functions using the test arguments in `btest.c`. You will get full credit for a puzzle if it passes all of the tests performed by `btest.c`, half credit if it fails one test, and no credit otherwise. (Note that `btest` will report a correctness score out of 36 points, which includes the 4 extra credit correctness points for `leastBitPos` and `bitCount`. To receive full credit for correctness, you only need to get 32 correctness points.)

Regarding performance, our main concern at this point in the course is that you can get the right answer. However, we want to instill in you a sense of keeping things as short and simple as you can. Furthermore, some of the puzzles can be solved by brute force, but we want you to be more clever. Thus, for each function we've established a maximum number of operators that you are allowed to use for each function. This limit is very generous and is designed only to catch egregiously inefficient solutions. You will receive two points for each function that satisfies the operator limit.

Finally, we've reserved 7 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

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.

Name	Description	Rating	Max Ops
`bitNor(x,y)`	`~(x|y)` using only `&` and `~`	1	8
`bitXor(x,y)`	`^` using only `&` and `~`	2	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
`bang(x)`	Compute `!x` without using `!` operator	4	12
`leastBitPos(x)`	Mark least significant 1 bit	Extra Credit	30
`bitCount(x)`	Count number of 1's in `x`	Extra Credit	40

Table 1: Bit-Level Manipulation Functions.

Function `bitNor` computes the NOR function. That is, when applied to arguments `x` and `y`, it returns `~(x|y)`. You may only use the operators `&` and `~`. Function `bitXor` should duplicate the behavior of the bit 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) 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 $1 \leq n \leq 31$. Function `bang` computes logical negation without using the `!` operator.

Extra Credit: 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. Function `bitCount` returns a count of the number of 1's in the argument.

Part II: Two's Complement Arithmetic

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)`	3	15
`abs(x)`	absolute value	4	10
`addOK(x,y)`	Does `x+y` overflow?	3	20

Table 2: Arithmetic Functions

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 $0 \leq n \leq 30$. It must round toward zero.

Function `abs` 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.

Checking Your Work

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

```
[you@attu3 lab1-handout]$ ./dlc bits.c
```

Type `./dlc -help` for a list of command line options. The README file is also helpful. Some notes on `dlc`:

- The `dlc` program runs silently unless it detects a problem.
- Don't include `<stdio.h>` in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages.

Check the file `README` for documentation on running the `btest` program. You'll 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, e.g., `./btest -f isPositive`.

Submitting Your Work

1. Make sure you have included your name in `bits.c`.
2. Remove any extraneous print statements.
3. Run `make submit` from the directory containing your copy of `bits.c`:

   ```
   [you@attu1 lab1-handout]$ make submit
   Zipping bits.c...
     adding: bits.c (deflated 63%)
   Zip succeeded.  Upload Lab1.zip to the assignment dropbox.
   ```

   This will create a zip file in the directory called `Lab1.zip`. Submit this file to the Catalyst dropbox for Lab 1 using the link on the CSE351 assignment web page.

If you discover a mistake and want to submit a revised copy, just re-upload your submission in Catalyst. *This will replace your previous submission.*