CS 213, Fall 1998
Homework Assignment H1
Assigned: Aug. 27, Due: Thurs., Sept. 10, 12:01AM

Randy Bryant (`Randy.Bryant@cs.cmu.edu`) is the lead person for this assignment.

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

You may work in a group of up to 2 people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory.

All files for this assignment are in the directory:

```
/afs/cs.cmu.edu/academic/class/15213-f98/H1
```

Start by copying the file `H1.tar` from that directory to a (protected) directory in which you plan to do your work. Then give the command: `tar xf H1.tar`. This will cause 6 files to be unpacked into the directory: `README`, `Makefile`, `bits.h`, `bits.c`, `btest.c`, `ftime.h`, and `ftime.c`. The only file you will be modifying and turning in is `bits.c`. You don't need to worry about the two "ftime" files at all. The file `btest.c` allows you to evaluate the correctness and performance of your code. The file `README` contains additional documentation. Use the command `make btest` to generate the test code and run it with the command `./btest`.

Looking at the file `bits.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. Do this right away so you don't forget. This file also contains function skeletons for each of the programming puzzles. You will insert your solution code into these functions.

Programming Rules

You must write your code in a highly stylized fashion. This is not intended to teach you good coding practice! Each function must be structured as shown in Figure 1. Each $Expr_i$ is an expression using only the following parts of C:

1. Long integer constants `0L` through `255L` (`0xFFL`), inclusive.

```
long int Funct( ... )
{
    /* Informative comment about code line below */
    long int var1 = Expr1;
    /* Informative comment about code line below */
    long int var2 = Expr2;
    ...
    /* Informative comment about code line below */
    long int vark = Exprk;
    /* Informative comment about code line below */
    return Exprr;
}
```

Figure 1: Required Coding Style

2. Unary long integer operations `!`[1] and `~`.
3. Casting from `int` to `long int` (either explicit or implicit.)
4. Binary long integer operations `&`, `^`, `|`, `+`, `<<`, and `>>`.
5. Calls to one of the other functions in the file [Not recommended for performance reasons, but useful for debugging].
6. Calls to functions of the form `test_`*Funct*, where *Funct* is one of the functions in this file [Only if you could not get a correct solution to *Funct*].

Some of the problems restrict the set of allowed operators even further.

You are expressly forbidden to:

1. Use any control constructs such as `if`, `do`, `while`, `switch`, `for`, etc.
2. Define or use any macros.
3. Define any additional functions in this file.
4. Call any functions outside this file [other than ones of the form `test_`*Funct*, where *Funct* is one of the functions in this file, but you were unable to get it working.]
5. Use any other operations, including `&&`, `||`, `-`, and `?:`.
6. Any use of explicit or implicit casting, other than from `int` to `long int`,

You may assume your machine:

1. Uses 2's complement representations of integers.
2. Performs right shifts arithmetically.

[1] Actually, `!` is a "predicate" operator. It can take a `long int` as an argument, but the result will be an `int`.

3. Uses either a 32- or 64-bit representation of long integers.

4. Has unpredictable behavior when shifting a long integer by more than the word size.

Evaluation

Your code will be tested on two different machines: one of our "colour" Alpha's, and an Andrew Sun. [There are many Sun machines in the clusters. Alternatively, you can reach a Sun server by telnet'ing to `sun4.andrew.cmu.edu`.] Your grade will be computed out of a maximum of 100 points based on the following distribution:

38 Correctness of code running on Alpha.

38 Correctness of code running on Sun.

19 Performance of code running on Alpha

5 Style points, based on a subjective evaluation of the quality of your solutions and your comments.

The 18 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 38. For correctness testing, we will first check that your code obeys the prescribed coding style. You will not get any correctness points for a function using an out-of-compliance coding style. Then we will evaluate your functions on a number of arguments (somewhat more than those in your file `btest.c`. You will get full credit for a puzzle if it passes all our tests, half credit if it fails one test, and no credit otherwise.

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. Thus, for each function we've established a maximum average time (expressed in clock cycles) that your function should take each time it is called, with the function call overhead subtracted off. Using as weight the difficulty rating of a problem divided by 2 (to give a maximum weighted sum of 19), we will give you credit as follows.

- No credit if an error is detected in the function.
- Full credit if your code runs within the allotted time.
- 2/3 credit if your function runs between 1.0X and 1.5X of the allotted time.
- 1/3 credit if your function runs between 1.5X and 2.0X of the allotted time.
- No credit if your function runs more than 2.0X of the allotted time.

We have set the limits fairly generously. It should not be difficult to stay within them.

Finally, we've reserved 5 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.

You'll find the puzzles cumulative. The solution to one will be useful in solving later ones. This could make life difficult if you are unable to solve one of the puzzles and hence cannot solve an entire sequence. To avoid getting overly penalized in this way, follow these rules if you are unable to get the code for some function *Funct* working:

Name	Args.	Description	Rating	Limit
`bitOr`	2	`|` using only `&` and `~`	1	1
`bitXor`	2	`^` using only `&` and `~`	2	2
`isZero`	1	`x == 0`	1	2
`isEqual`	2	`x == y`	2	2
`anyMaskedBits`	2	Check for any masked bits	2	3
`allMaskedBits`	2	Check for all masked bits	2	3

Table 1: Bit-Level Manipulation Functions

Name	Args.	Description	Rating	Limit
`is32`	0	32-bit long integers?	2	1
`signBit`	0	Position of long integer sign bit	3	1
`highByte`	1	Most significant byte of `x`	3	2

Table 2: Word-Size Dependent Functions

1. Do the best you can. You will be penalized both correctness and performance points for this function.
2. In other parts of your file, you may call the function `test_`*Funct*. This is a test function from the file `btest.c` designed to match the behavior of *Funct*.
3. You will not be penalized any correctness points for calling `test_`*Funct* in place of an erroneous *Funct*.

Part I: Bit manipulations

Table 1 describes a set of functions that manipulate and test sets of bits. As indicated by the "Args." field, each function takes either a single long integer argument `x` or two long integer arguments `x` and `y`. The "Rating" field gives the difficulty rating for the puzzle, and the "Limit" field gives the performance time limit.

Functions `bitOr` and `bitXor` should duplicate the behavior of the bit operations `|` and `^`, respectively. In these, you may only use the operations `&` and `~`.

Functions `isZero` and `isEqual` perform comparisons of `x` to 0 and to `y`, respectively. As with all of our "predicate" operations, they should return `1L` if the tested condition holds and `0L` otherwise. [Note that this is a slight departure from C semantics—technically all predicate operators in C return integers rather than long integers.]

Functions `anyMaskedBits` and `allMaskedBits` both treat argument `y` as a "mask." That is, each bit position where the mask is set to 1 indicates an "interesting" bit position. The functions consider the bit values of `x` at these bit positions. Function `anyMaskedBits` determines whether `x` has a one at any mask bit position, while `allMaskedBits` determines whether `x` has a one at all of these bit positions.

Part II: Word Size Dependencies

Name	Args.	Description	Rating	Limit
`TMin`	0	smallest long integer	1	3
`minusOne`	0	`-1L`	1	1
`TMax`	0	largest long integer	1	3
`isNegative`	1	`x < 0L`	2	2
`isPositive`	1	`x > 0L`	2	3
`negate`	1	`-x`	2	3
`absval`	1	`x < 0 ? -x : x`	4	3
`isGreater`	2	`x > y`	3	9
`bang`	1	`!x` using the rest	4	6

Table 3: Arithmetic Functions

Table 2 describes a set of functions that critically depend on the word size of the machine. You must write code that works correctly on both 32- and 64-bit machines.

Function `is32` takes no arguments. It simply indicates whether the machine executing the program uses 32-bit long integers. Similarly, function `signBit` indicates the bit position of the sign bit on the machine, where the least significant bit is at position 0, while the most is at position 31 or 63.

Function `highByte` extracts the most significant byte from `x` and returns a result with this byte at the least significant bit positions and all other bits set to 0.

Part III: Two's Complement Arithmetic

Table 3 describes a set of functions that make use of the two's complement representation of integers.

Functions `TMin`, `minusOne`, and `TMax` return the smallest representable long integer, -1, and the largest representable long integer, respectively, for the machine executing the program.

Functions `isNegative`, and `isPositive` determine whether `x` is less than or greater than 0, respectively, while `negate` computes `-x`. Function `absval` duplicates the behavior of the C code `(x < 0 ? -x : x)`. For almost all arguments, this will give the absolute value. For the minimum two's complement long integer T_{min}, however, this will simply be an identity operation.

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

Finally, function `bang` should duplicate the behavior of the C expression `!x`. You may use any of the allowed operators except for `!` in your solution. However, you may not call any functions within the body of this procedure. This restriction is imposed since some of the other procedures may make use of the `!` operator.

Advice

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 isNegative
```

Hand In

Make sure you have included your identifying information in your file `bits.c`. Remove any extraneous print statements. Make a copy of your file named `bits-`*aid1*`.c`, where *aid1* is the Andrew Id of the first person on your team. Copy this file to the directory

```
/afs/cs.cmu.edu/academic/class/15213-f98/H1/handin
```

You only have write permission to this directory. If you make a mistake, e.g., by giving the file an incorrect name, send a copy of the correct file via email to the contact person for this assignment along with an explanation.

You may only hand in an assignment once. Make sure your code is in its final form before handing it in.