CSE 351 Winter 2011
HW1: Manipulating Bits (using C)
Assigned: Friday January 7
Due: Friday January. 14, 11:59PM

1 Overview and Goal

The purpose of this assignment is to become more familiar with data at the bit-level representation. 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.

2 Instructions

You can fetch the files required for this homework at
`http://www.cs.washington.edu/education/courses/cse351/11wi/homework/hw1Files/hw1.tar.gz`
(Un-tar'ing will create a `hw1` directory. Refer to the instructions for HW0 if you don't remember what to do with a `.tar.gz` file.)

The distribution contains a number of tools, described later, and a `bits.c` file. `bits.c` contains a skeleton for each of the programming puzzles, along with a large comment block that describes exactly what the function must do and what restrictions there are on its implementation. Your assignment is to complete each function skeleton using only *straightline* code (i.e., no loops or conditionals) and a limited number of C arithmetic and logical operators. Also, you are not allowed to use any constants longer than 8 bits. The intent of the restrictions is to require you to think about the data as bits - because of the restrictions, your solutions won't be the most efficient way to accomplish the function's goal, but the process of working out the solution should make the notion of data as bits completely clear.

Note: *The infrastructure for this assignment may require a 32-bit system - it tries to force gcc to generate a 32-bit executable, but I can't guarantee it works. If you use a 64-bit system, verify that your code works on a 32-bit system before turn-in (e.g., using attu.cs).*

3 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`. More complete (and definitive, should there be an inconsistencies) documentation is found in the `bits.c` file itself.

3.1 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. See the comments in `bits.c` for more details on the desired behavior of the functions. You may also refer to the test functions in `tests.c`. These are used as reference functions to express the correct behavior of your functions, although they don't satisfy the coding rules for your functions.

Name	Description	Rating	Max Ops
`bitNor(x,y)`	`x nor y` using only `&` and `~`	1	8
`bitXor(x,y)`	`x ^ y` using only `&` and `~`	1	14
`thirdBits()`	returns a word with every third bit set	1	8
`isNotEqual(x,y)`	return 0 if `x == y`, and 1 otherwise	2	6
`byteSwap(x,n,m)`	swaps the `n`th byte and the `m`th byte	2	25
`logicalShift(x,n)`	shift `x` to the right by `n`, using a logical shift	3	20
`isAsciiDigit(x)`	return 1 if `0x30 <= x <= 0x39`	3	15
`conditional(x,y,z)`	same as `x ? y : z`	3	16
`bang(x)`	Compute `!n` without using `!` operator.	4	12

Table 1: Bit-Level Manipulation Functions.

3.2 Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers. Again, refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

Name	Description	Rating	Max Ops
`minusOne()`	return a value of -1	1	2
`negate(x)`	return `-x`	2	5
`isPositive(x)`	return 1 if `x > 0`, 0 otherwise	3	8
`isPower2(x)`	returns 1 if `x` is a power of 2, and 0 otherwise	4	20

Table 2: Arithmetic Functions

Checking your work

We have included some tools to help you check the correctness of your work.

- **`dlc`:** This is a modified version of an ANSI C compiler from the MIT CILK group that you can use to check for compliance with the coding rules for each puzzle. The typical usage is:

  ```
  $ ./dlc bits.c
  ```

  The program runs silently unless it detects a problem, such as an illegal operator, too many operators, or non-straightline code in the integer puzzles. Running with the `-e` switch:

  ```
  $ ./dlc -e bits.c
  ```

  causes `dlc` to print counts of the number of operators used by each function. Type `./dlc -help` for a list of command line options.

- **`btest`:** This program checks the functional correctness of the code in `bits.c`. To build and use it, type the following two commands:

  ```
  $ make
  $ ./btest
  ```

  Notice that you must rebuild `btest` each time you modify your `bits.c` file.

  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:

  ```
  $ ./btest -f bitNor
  ```

  You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`:

  ```
  $ ./btest -f bitNor -1 7 -2 0xf
  ```

  Check the file `README` for documentation on running the `b`test program.

4 Turn-in

Turn-in is online. Submit just your `bits.c` file.

5 Advice

- Do not include the `<stdio.h>` header file in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages. You will still be able to use `printf` in your `bits.c` file for debugging without including the `<stdio.h>` header, although `gcc` will print a warning that you can ignore.
- You should be able to use the debugger on your code. For example:

```
$ make
gcc -O -Wall -m32 -g -lm -o btest bits.c btest.c decl.c tests.c
gcc -O -Wall -m32 -g -o fshow fshow.c
gcc -O -Wall -m32 -g -o ishow ishow.c
$ gdb ./btest
GNU gdb (GDB) 7.2-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/zahorjan/courses/cse351/11wi/homework/hw1/hw1-handout/bte
(gdb) b bitNor
Breakpoint 1 at 0x8048717: file bits.c, line 144.
(gdb) r
Starting program: /home/zahorjan/courses/cse351/11wi/homework/hw1/hw1-handout/btest
Score Rating Errors Function

Breakpoint 1, bitNor (x=-2147483648, y=-2147483648) at bits.c:144
144 }
(gdb) p x
$1 = -2147483648
(gdb) p/x x
$2 = 0x80000000
(gdb) q
A debugging session is active.

 Inferior 1 [process 12728] will be killed.

Quit anyway? (y or n) y
$
```

- The `dlc` program enforces a stricter form of C declarations than is the case for C++ or that is enforced by `gcc`. In particular, any declaration must appear in a block (what you enclose in curly braces) before any statement that is not a declaration. For example, it will complain about the following code:

```
int foo(int x)
```

```
{
  int a = x;
  a *= 3;     /* Statement that is not a declaration */
  int b = a;  /* ERROR: Declaration not allowed here */
}
```