

Datorarkitektur, VT 2009

Lab 1: Manipulating Bits

Inge Frick

2008-12-30

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

You may work in a group of up to two 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 the course Web page.

Hand Out Instructions

Start by copying `/info/maskin09/labbar/lab1/datalab-handout.tar` to a (protected) directory in which you plan to do your work. Then give the command:

```
tar xvf datalab-handout.tar.
```

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

The file `btest.c` allows you to evaluate the functional correctness of your code. The file `README` contains additional documentation about `btest`. Use the command

```
make btest
```

to generate the test code and run it with the command

```
./btest.
```

The remaining files are used to build `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.

The `bits.c` file also contains a skeleton for each of the 15 programming puzzles. 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. 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.

Evaluation

Your code will be compiled with `GCC` and run and tested on one of CSCs Unix machines. Your score will be computed out of a maximum of 75 points based on the following distribution:

- 40 Correctness of code running on one of CSCs Unix machines.
- 30 Performance of code, based on number of operators used in each function.
- 5 Style points, based on a subjective evaluation of the quality of your solutions and your comments.

The 15 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 40. 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.

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 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.

Based on these points you will get a grade on this work. 40 points will give grade E, 49 points will give grade D, 57 points will give grade C, 65 points will give grade B and 73 points will give grade A.

Part I: Two's Complement Arithmetic

Name	Description	Rating	Max Ops	Record
<code>minusOne(void)</code>	return a value of -1	1	2	1
<code>fitsBits(x,n)</code>	can x be represented as n-bit integer?	2	15	6
<code>isLessOrEqual(x,y)</code>	$x \leq y$?	3	24	10
<code>isLess(x,y)</code>	$x < y$?	3	24	10
<code>sm2tc(x)</code>	Convert from sign-magnitude to two's complement	4	15	5

Table 1: Arithmetic Functions

Table 1 describes a set of functions that make use of the two's complement representation of integers.

Function `minusOne` returns two's complement `-1`.

Function `fitsBits` determines whether `x` can be represented as an `n`-bit, two's complement integer.

Function `isLessOrEqual` determines whether `x` is less than or equal `y`.

Function `isLess` determines whether `x` is less than `y`.

Function `sm2tc` Convert from sign-magnitude (where the MSB is the sign bit) to two's complement.

Part II: Bit manipulations

Name	Description	Rating	Max Ops	Record
<code>bitAnd(x,y)</code>	$(x \& y)$ using only <code> </code> and <code>~</code>	1	8	4
<code>isZero(x)</code>	$x = 0?$	1	2	1
<code>isEqual(x,y)</code>	$x = y?$	2	5	2
<code>evenBits(x,n)</code>	return word with even-numbered bits set to 1	2	8	4
<code>bitMask(hb,lb)</code>	Generate mask with of 1's between <code>lb</code> and <code>hb</code>	3	16	5
<code>reverseBytes(x)</code>	reverse the bytes of <code>x</code>	3	25	12
<code>conditional(x,y,z)</code>	$x ? y : z$	3	16	6
<code>reverseBits(x)</code>	reverse the bits of <code>x</code>	4	90	34
<code>isPower2(x)</code>	returns 1 if <code>x</code> is a power of 2, and 0 otherwise	4	60	6
<code>isNonZero(x)</code>	$x \neq 0?$	4	10	5

Table 2: Bit-Level Manipulation Functions.

Table 2 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, the “Max ops” field gives the maximum number of operators you are allowed to use to implement each function, and the “Record” field gives the minimum number of operators needed as we know it.

Function `bitAnd` computes the AND function. That is, when applied to arguments `x` and `y`, it returns $(x \& y)$. You may only use the operators `|` and `~`.

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

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

Function `evenBits` Returns a word with even-numbered bits set to 1 and odd-numbered bits set to 0.

Function `bitMask` Assume $0 \leq lb \leq 31$, and $0 \leq hb \leq 31$ If $lb > hb$, then return 0 else return a word where bits with, $lb \leq position \leq hb$, set to 1 and others set to 0.

Function `reverseBytes` Return a word with reversed bytes.

Function `conditional` is equivalent to $x ? y : z$.

Function `reverseBits` Return a word with all bits in reversed order.

Function `isPower2` Returns 1 if `x` is a power of 2, and 0 otherwise.

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

Advice

You are welcome to do your code development using any system or compiler you choose. Just make sure that the version you turn in compiles and runs correctly on our Solaris computers. If it doesn't compile, it can't be graded.

The `d1c` program, a modified version of an ANSI C 88 compiler, will be used to check your programs for compliance with the coding style rules. `d1c` is available on Solaris computers when you have logged in to the course. The typical usage is

```
d1c bits.c
```

Type `d1c -help` for a list of command line options. The README file is also helpful. Some notes on `d1c`:

- The `d1c` program runs silently unless it detects a problem.
- Don't include `<stdio.h>` in your `bits.c` file, as it confuses `d1c` and results in some non-intuitive error messages.
- Note that `d1c` as a ANSI C 88 compiler doesn't allow mixing of declarations and statements. Each `{ }` block starts with declarations and then comes the statements.

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`.

Hand In Instructions

- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- Create a team name of the form:
 - “*ID*” where *ID* is your CSC user ID, if you are working alone, or
 - “*ID*₁+*ID*₂” where *ID*₁ is the CSC user ID of the first team member and *ID*₂ is the CSC user ID of the second team member.

This should be the same as the team name you entered in the structure in `bits.c`.

- To handin your `bits.c` file, type:

```
make handin TEAM=teamname
```

where `teamname` is the team name described above.

- After the handin, if you discover a mistake and want to submit a revised copy, type

```
make handin TEAM=teamname VERSION=2
```

Keep incrementing the version number with each submission.

- You can verify your handin by looking in

```
/info/maskin09/labbar/lab1/handin
```

You have list and insert permissions in this directory, but no read or write permissions.