Datalab

Due: Wednesday, October 5, 11:59 PM

Please submit your `bits.c` to Bruinlearn lab1 assignment here

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.

0. Download and Files

Download lab files here.

The (strongly) recommended approach is to use a lab machine for programming and testing (preferably `cs33.seas.ucla.edu`). See the the resources page for help.

If you are logged-in to a lab machine through the command line, you can use `wget` to download:

```
   unix> wget https://polyarch.github.io/cs33/labs/datalab-handout.tar
```

Then use `tar` to unpack: (This will cause a number of files to be unpacked in the directory.)

```
   unix> tar -xvf datalab-handout.tar
```

The only file you will be modifying and turning in is `bits.c`. **Your goal is to modify your copy of bits.c so that it passes all the tests in btest without violating any of the coding guidelines.**

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 file dlc is a compiler binary that you can use to check your solutions for compliance with the coding rules. The remaining files are used to build btest.

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

File Description:

- *Makefile*: Makes btest, fshow, and ishow
- *README*: This file
- *bits.c*: The file you will be modifying and handing in
- *bits.h*: Header file
- *btest.c*: The main btest program
    - *btest.h*: Used to build btest
    - *decl.c*: Used to build btest
    - *tests.c*: Used to build btest
- *dlc**: Rule checking compiler binary (data lab compiler)
- *ishow.c*: Utility for examining integer representations

1. Modifying bits.c and checking it for compliance with dlc

IMPORTANT: Carefully read the instructions in the `bits.c` file before you start. These give the coding rules that you will need to follow. Your bits.c file MUST comply with the coding guidelines.

Use the dlc compiler (./dlc) to automatically check your version of `bits.c` for compliance with the coding guidelines:

```
  unix> ./dlc bits.c
```

dlc returns silently if there are no problems with your code. Otherwise it prints messages that flag any problems. Running dlc with the -e switch:

```
unix> ./dlc -e bits.c
```

causes dlc to print counts of the number of operators used by each function. Running dlc with the -e flag should always give you outputs in the terminal. If you see a blank here, be careful - your `./dlc` is likely corrupted! Please see the FAQs section below for more details.

Please note that dlc requires you to separate out the variable declarations from the rest of the code, placing the declarations first. I.e. place ALL variable declarations first before any other lines of code. If you do not it may mistakenly cause a parse error.

Once you have a legal solution, you can test it for correctness using the ./btest program.

2. Testing with btest

The Makefile in this directory compiles your version of `bits.c` with additional code to create a program (or test harness) named btest.

To compile and run the btest program, type:

```
unix> make btest
unix> ./btest [optional cmd line args]
```

You will need to recompile btest each time you change your `bits.c` program. When moving from one platform to another, you will want to get rid of the old version of btest and generate a new one. Use the commands:

```
unix> make clean
unix> make btest
```

btest tests your code for correctness by running millions of test cases on each function. It tests wide swaths around well known corner cases such as Tmin and zero for integer puzzles, and zero, inf, and the boundary between denormalized and normalized numbers for floating point puzzles. When btest detects an error in one of your functions, it prints out the test that failed, the incorrect result, and the expected result, and then terminates the testing for that function.

Here are the command line options for btest:

```
unix> ./btest -h
Usage: ./btest [-hg] [-r <n>] [-f <name> [-1|-2|-3 <val>]*] [-T <time limit>]
  -1 <val>  Specify first function argument
  -2 <val>  Specify second function argument
  -3 <val>  Specify third function argument
  -f <name> Test only the named function
  -g        Format output for autograding with no error messages
  -h        Print this message
  -r <n>    Give uniform weight of n for all problems
  -T <lim>  Set timeout limit to lim
```

Examples:

Test all functions for correctness and print out error messages:

```
unix> ./btest
```

Test all functions in a compact form with no error messages:

```
unix> ./btest -g
```

Test function foo for correctness:

```
unix> ./btest -f foo
```

Test function foo for correctness with specific arguments:

```
unix> ./btest -f foo -1 27 -2 0xf
```

btest does not check your code for compliance with the coding guidelines. Use dlc to do that. So please run both ./dlc and ./btest, one does not replace the other.

3. Grading

The score will be computed on a total of 41 points based on the following distribution:

19 Correctness points. 18 Performance points. 4 Style points.

Correctness points - The puzzle questions have been given a difficulty rating between 1 and 3, such that their weighted sum totals to 19. We will evaluate your functions using the btest program, which is described in the next section. You will get full credit for a puzzle if it passes all of the tests performed by btest, and no credit otherwise.

Performance points - 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 2 points for each correct function that satisfies the operator limit.

Style points - Finally, we've reserved the style 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.

Extra Credit

Also, we have included two extra tricky puzzle questions for extra credits: `leastBitPos` and `greatestBitPos`. Each question counts for 2 extra points: 1 correctness point and 1 performance point. There are 4 points of extra credits in total. You will get extra credit (up to 4 points) if your total lab score is more than 41 points.

4. Helper Programs

We have included the ishow program to help you decipher integer representations. It takes a single decimal or hex number as an argument. To build it type:

```
unix> make ishow
```

Example usages:

```
unix> ./ishow 0x27
Hex = 0x00000027, Signed = 39, Unsigned = 39

unix> ./ishow 27
Hex = 0x0000001b, Signed = 27, Unsigned = 27
```

5. Frequently Asked Questions

I am receiving this error message when I try to use the data lab compiler to validate my program, is this normal?

```
./dlc: /lib64/libc.so.6: version `GLIBC_2.14' not found (required by ./dlc```)
```

This means that you don't have the right library on your machine – try using `cs33.seas.ucla.edu` instead.

Everything passes in ./btest just fine, yet `./dlc` keeps throwing an error that doesn't make sense to me

```
bits.c:242: parse error

bits.c:246: undeclared variable `sign'

bits.c:246: undeclared variable `remainder'
```

As mentioned in the description, `dlc` is kind-of picky – it requires all variable declarations (eg. `int a,b,c;` etc.) before any other lines of code, so just move those first.

I always get this warning when running `./dlc`, is that okay?

```
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included
from includable file /usr/include/stdc-predef.h.
```

Yes, getting that warning is okay.

Is it okay that dlc -e returns without printing anything?

This is the typical output for `./dlc -e bits.c` (with different line number and operator count)

```
dlc:bits.c:169:tmax: 0 operators
dlc:bits.c:179:isZero: 0 operators
dlc:bits.c:189:bitXor: 0 operators
dlc:bits.c:199:isNotEqual: 0 operators
```

```
dlc:bits.c:210:sign: 0 operators
dlc:bits.c:220:conditional: 0 operators
dlc:bits.c:232:replaceByte: 0 operators
dlc:bits.c:244:isAsciiDigit: 0 operators
dlc:bits.c:255:subtractionOK: 0 operators
dlc:bits.c:268:leastBitPos: 0 operators
dlc:bits.c:279:greatestBitPos: 0 operators
```

Please be suspicious if dlc returns silently! Occasionally, a student's `./dlc` becomes replaced with an empty file (I don't know how).

In that case, dlc may still run but return silently and **not check your code** – yikes! You can check if it's empty by running `ls -alh ./dlc`, and look at the file size. If it is empty, please download a new copy of the lab and run with that new copy.

How do I debug with gdb? It's not showing any symbols?

You'll need to compile with the debug flag, ie. `-g`. To do this, open up your `makefile`, and add `-g` to the list of flags like this:

```
CFLAGS = -O -Wall -m32 -g
```