CSE 30, Fall 2016
Lab 01: Data Lab: Manipulating Bits

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers and floating point numbers. 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 Logistics

If you are working in a pair you must collaborate with your partner following the pair programming guidelines for the course. Otherwise, you must work individually and adhere to the Academic Integrity rules of the course.

3 Obtaining the starter code

Click on this link to create your repo and get the starter code for the assignment:

Get Lab01 on github classroom

If clicking on the link doesn't work, copy and paste the url on your browser:

https://classroom.github.com/group-assignment-invitations/9e12c7be53fea641397bd691db5e1307

If you are working with a partner, both of you will have to click on the above link. However, only one of the partners have to create a joint repo. The other partner only needs to join the repo following the instructions provided by the above link.

If you are working in a pair create your repo following the naming convention:

```
<partner1-firstname-lastinitial>-<partner2-firstname-lastinitial>
```

For example if the partners were Joe Smith and Lily Monroe, the naming convention should be JoeS-LillyM. Note that the partner whose first name appears earlier lexicographically should appear first in this naming convention. Choosing LillyM-JoeS as the repo name would be incorrect in this case

If you are working individually follow the naming convention:

```
<partner1-firstname-lastinitial>
```

Once you have create a new repo, or joined an existing repo, you will have access to the starter code and you can start working on the assignment.

You ONLY need to modify the files `bits.c`, `bits_ARM.c`, `test_bits.c` `test_bits_ARM.c`

You have to turn in all your files and on github. In addition, you should turn in ONLY `bits.c` and `bits_ARM.c` via gradescope.

The `bits.c` file contains a skeleton for each of the 15 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code for the integer puzzles (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:

```
 ! ˜ & ˆ | + << >>
```

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 `bits_ARM.c` requires that you compile by hand your implementation of a subset of the functions that you implemented in bits.c. This means that you have to write ARM assembly without the aid of the compiler, using ONLY straight line code.

4 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`.

4.1 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 `test_bits.c`. The test file has some test code that you should build on to test your functions in bits.c.

4.2 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.

Implement the ARM Assembly verison of all a subset of the above functions in bits_ARM.s. Specifically you are required to implement the functions:

Name	Description	Rating	Max Ops
`bitAnd(x,y)`	`x & y` using only `|` and `~`	1	8
`getByte(x,n)`	Get byte `n` from `x`.	2	6
`logicalShift(x,n)`	Shift right logical.	3	20
`bitCount(x)`	Count the number of 1's in `x`.	4	40
`bang(x)`	Compute `!n` without using `!` operator.	4	12

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`tmin()`	Most negative two's complement integer	1	4
`fitsBits(x,n)`	Does `x` fit in `n` bits?	2	15
`divpwr2(x,n)`	Compute $\mathtt{x}/2^{\mathtt{n}}$	2	15
`negate(x)`	`-x` without negation	2	5
`isPositive(x)`	`x > 0`?	3	8
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`ilog2(x)`	Compute $\lfloor \log_2(\mathtt{x}) \rfloor$	4	90

Table 2: Arithmetic Functions

bitAnd_ARM getByte_ARM logicalShift_ARM bitCount_ARM fitsBits_ARM negate_ARM isLessOrEqual_ARM

Each of the above functions should behave in the same way as the C equivalent. We will however not deduct points if your ARM functions don't meet the constraints posed on the C functions on the number and type of operators used. However, your ARM functions should be a direct translation of your C functions as far as possible.

We have provided the skeleton code for two ARM functions in `bits_ARM.s`, you should complete these two functions and add code for all the other remaining functions. Test your assembly code by writing test functions in C in the file `test_bits_ARM.c`

4.3 Floating-Point Operations

For this part of the assignment, you will implement some common single-precision floating-point operations. In this section, you are allowed to use standard control structures (conditionals, loops), and you may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function as having type `unsigned`, and any returned floating-point value will be of type `unsigned`. Your code should perform the bit manipulations that implement the specified floating point operations.

Table 3 describes a set of functions that operate on the bit-level representations of floating-point numbers. Refer to the comments in `bits.c`

Name	Description	Rating	Max Ops
`float_neg(uf)`	Compute `-f`	2	10
`float_i2f(x)`	Compute `(float) x`	4	30
`float_twice(uf)`	Computer `2*f`	4	30

Table 3: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

Functions `float_neg` and `float_twice` must handle the full range of possible argument values, including not-a-number (NaN) and infinity. The IEEE standard does not specify precisely how to handle NaN's. We will follow a convention that for any function returning a NaN value, it will return the one with bit representation `0x7FC00000`.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, switch to the handout directory and type:

```
unix> make
```

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
unix> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

You can also give `fshow` hexadecimal and floating point values, and it will decipher their bit structure.

We would highly recommend that you use the test suite provided to you to come up with your own test code. For now make sure your testcode behaves similarly to btest. This should give you practice with developing your own test code which is something you will have to do in later assignments.

5 Evaluation

Your score will be computed out of a maximum of 76 points based on the following distribution:

41 Correctness points (for bits.c and bits_ARM.s)

30 Performance points (only for bits.c)

5 Style points.

Correctness points. The 15 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 41. We will evaluate your functions using the our gradescope autograder.

You will get full credit for a puzzle if it passes all of the tests performed on gradescope. Code that doesn't compile will receive 0 points. You may submit your code on gradescope as many times as you like.

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 two points for each correct function that satisfies the operator limit for your solutions in bits.c. We are not imposing a similar restriction on your ARM code, but highly recommend that you write the shortest and most elegant ARM code that follows your C implementation.

Style points. 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.

Autograding your work

We have included an autograding tool in the starter code to check for performance of bits.c —`dlc` 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:

```
unix> ./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:

```
unix> ./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.

Note that our gradescope autograder does not run dlc on your code. You should run this individually to make sure it complies with the constraints for each function. We will run the dlc compiler on your submission after the submission deadline.

6 Handin Instructions

To make a submission you must do the following steps

1. commit and push your code on github following these instructions. Your code must include the completed versions of bits.c bits_ARM.s test_bits.ctest_bits_ARM.s in addition to all of the other skeleton code provided to you.

```
$ git add .
$ git commit -m "Final Submission"
$ git push origin master
```

2. Submit ONLY your `bits.c` solution file to the PA1 assignement on gradescope.

3. Submit ONLY you `bits_ARM.s` solution to the PA1-ARM assignment on gradescope

You may submit as many times as you want on gradescope before the deadline

7 Advice

- Don't 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.
- 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 */
}
```

8 Acknowledgements

I would like to thank Randal E. Bryant and David R'Hallaron at CMU for the original version of this assignment