Homework 4

System Skill (Term III/2022)

built on 2023/07/03 at 00:26:57 ***due:*** *Thursday, July 13th @ 11:59pm*

This assignment will give you practice with manipulating bits, two's complement integers, and basic assembly. You will write some code and hand it in.

Hand-in Instructions

To submit this assignment, please follow the steps below:

1. Because syskill is currently down for the time being, submit your zip file on canvas.
2. Make sure you install 32-bit library for GCC. On Ubuntu and WSL, this can be done by

   ```
   sudo apt-get install gcc-multilib
   ```

3. The files to hand in are

   ```
   bits.c
   disas.c
   ```

   Zip up all the files as `a4.zip`
4. Find out the MD5 hash of your zip file. You will need to submit this code on Canvas. We use it for keeping track of your submission time. You may resubmit your work but the MD5 hash has to match.

   ```
   > md5sum a4.zip
   ```

5. Log on to Canvas, go to assignment 4submission page, and enter the MD5 hash and your zip file.

1 Datalab (70 points)

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.

1.1 Handout Instructions

You will need a starter package, which ships in the form of a zip file.

After unzip, 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` inside `datalab` directoty.

The `bits.c` file contains a skeleton for each of the 10 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:

```
! ~ & ^ | + << >>
```

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.

1.2 The Puzzles

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

1.2.1 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
`minusOne(void)`	Return the value -1	1	2
`sign(x)`	Return 1 if positive, 0 if zero, -1 if negative	2	10
`anyEvenBit(x)`	Return 1 if any even bit in `x` is set to 1	2	12
`getByte(x,n)`	Extract byte n from word x	2	6
`negate(x)`	Compute `-x`	2	5
`byteSwap(x,n,m)`	Swap the `m` and `n` byte of `x`	2	25
`ezThreeFourths(x)`	Multuply x by 3/4 and round it toward 0	3	12
`bang(x)`	Compute `!n` without using `!` operator.	4	12
`ilog2(x)`	Return $log_2(x)$	4	90
`greatestBitPos(x)`	Compute a mask marking the most significant 1 bit.	4	70
`isPower2(x)`	Return 1 if `x` is a power of 2	4	20
`absVal(x)`	Compute the absolute value of `x`	4	10
`bitCount(x)`	Count the number of 1's in `x`.	4	40

Table 1: Bit-Level Manipulation Functions.

1.2.2 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
`tmin(void)`	Return minimum number	1	4
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`isPositive(x)`	`x > 0`?	3	8

Table 2: Arithmetic Functions

1.2.3 Floating Point Arithmetic

Table 3 describes a set of functions that make use of operations on floating point values. Again, refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

Name	Description	Rating	Max Ops
`float_abs(uf)`	Compute absolute of `uf`	2	10
`float_i2f(x)`	Convert `x` to float	4	30
`float_twice(uf)`	Compute `2*uf`	4	30

Table 3: Floating Point Functions

1.3 Evaluation

Your score will be computed by multiplying the rating points with 2 out of a maximum of 70 points. Any points extra (up to 78 is your extra credit). Only answer that are correct and performs well (defined below) count.

Correctness points. The 19 puzzles you must solve have been given a difficulty rating between 1 and 4. A difficulty rating of 1 equals is worth 1 point, a rating of 2 is worth 2 points, a rating of 3 is worth 8 points, and a rating of 4 equals is worth 12 points. 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.

Scoring. The total points you can get is 70. Any excess points will be converted to EXTRA CREDIT at the conversion rate of 25%.

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.

Autograding your work

We have included some autograding tools in the handout directory — `btest` and `dlc` — to help you check the correctness of your work.

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

```
unix> make
unix> ./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:

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

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

```
unix> ./btest -f bitAnd -1 7 -2 0xf
```

Check the file `README` for documentation on running the btest program.

- **`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.

1.4 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 */
  }
  ```

2 Basic In-line Assembly (30 points)

In many cases, you might need to write assembly as a part of your C/C++ program in order to take an even-lower control of your machine. In this question, you will implement two functions `void string_copy(char source, char* destination)` and `int string_compare(char* string1, char* string2` using in-line assembly through `asm`.

To do this, our starter code `asm.c` has the two function definition as well as some basic test code in the main function. You can assume that all the character arrays are already allocated and you have enough space for the two functions to perform their task.

`void string_copy(char* source, char* destination)` takes in two character arrays, and will copy the content of `source` to the `destination`. You need to make sure that the function puts the NULL character (i.e.,
0) at the end of the destination string. **Hint:** make use of the `mov` instruction and ensure that you move one byte at a time from one memory location to a temp register, then from a temp register to the target address.

`int string_compare(char* string1, char* string2` compares the two input strings and return a negative number if `string1` is lower in the alphabetically-sorted order than `string2`, positive number if `string1` is higher in the alphabetically-sorted order than `string2`, and zero if both strings are the same. **Hint:** this basically asks you to compare each characters until the cmp functions set the non-zero status (i.e., the zero bit is not set), and return negative number of the signed bit is set and positive number of the signed bit is not set.

To do this, you will take advantage of the extended `asm`, which allow you to put in the x86 assembly code at the location. The extended asm also allows you to map the variables to x86 registers of your choice. You can check out the documentation at `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`.

Example code at `https://ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO` can be helpful.

Note: You will also see how I use the “volatile” keyword. Volatile marks that the variable/code region can change unexpectedly from an external source. By doing this, codes that are involved with the volatile variables are considered "not safe" for compiler optimization that moves your compiled assembly code around. In order work, this will force the compiler not to do any optimization, transform the code involved with volatile, or move your volatile code around. Other common use of volatile are in embedded systems, which can put the volatile flag to variables that might get updated by external sources (such as temperature sensors, depth sensors, accelerometers, gyroscope readings, etc.).