built on 2017/02/24 at 14:46:39 ***due:*** *fri, mar 3 @ 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. The files to hand in are

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

   Zip up all the files as `a4a.zip`

2. 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 a4a.zip
   ```

3. Copy the zip file to the directory `/subm/u5712345` where `u5712345` is your student ID.

   ```
   > cp a4a.zip /subm/u5712345
   ```

4. Log on to Canvas, go to assignment 4asubmission page and enter the MD5 hash.

Task 1: Datalab (47 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.

For this lab to work properly, you'll need a Linux system. We recommend using the Hamachi machine.

Handout Instructions

You will need a starter package, which ships in the form of a tar file. On Hamachi, you can find `datalab-handout.tar` at

```
/courses/filepro-t2-2016/datalab-handout.tar
```

Start by copying `datalab-handout.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> 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 `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.

The Puzzles

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

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.

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
`conditional(x,y)`	Implement `x ? y : z`	3	16
`allEvenBits(x)`	Is all even-numbered bits in word set to 1?	2	12
`implication(x,y)`	Implement $x \implies y$	2	5
`bitCount(x)`	Count the number of 1's in `x`.	4	40
`bang(x)`	Compute `!n` without using `!` operator.	4	12
`logicalShift(x,n)`	Shift right logical.	3	20
`byteSwap(x,n,m)`	Swaps the n-th byte and the m-th byte	2	25
`leastBitPos(x)`	Compute a mask marking the least significant 1 bit.	2	6

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`tmin()`	Most negative two's complement integer	1	4
`isLessOrEqual(x,y)`	`x <= y`?	3	24

Table 2: Arithmetic Functions

Evaluation

Your score will be computed out of a maximum of 47 points based on the following distribution:

22 Correctness points.

20 Performance points.

5 Style points.

Correctness points. The 10 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 `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 two points for each correct function that satisfies the operator limit.

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 some autograding tools in the handout directory — `btest`, `dlc`, and `driver.pl` — 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.

- **`driver.pl`:** This is a driver program that uses `btest` and `dlc` to compute the correctness and performance points for your solution. It takes no arguments:

  ```
  unix> ./driver.pl
  ```

  Your instructors will use `driver.pl` to evaluate your solution.

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 */
  }
  ```

Task 2: Basic disassembly (15 points)

Your final task is to pick apart a binary-encoded x86-64 machine instruction and print its human-readable assembly equivalent. This is the job of a *disassembler* (the inverse of the assembler), such as the `objdump` tool or the gdb `disassemble` command. The instructions you must decode are the five variants of the `pushq` instruction shown in the table below.

Disassembled instruction	Variant (what is pushed?)	Binary-encoded machine instruction (num bytes in parens)	In hex
`pushq $0x3f10`	immediate constant	`01101000 00010000 00111111 00000000 00000000` (5)	`68 10 3f 00 00`
`pushq %rbx`	register	`01010011` (1)	`53`
`pushq (%rdx)`	indirect	`11111111 00110010` (2)	`ff 32`
`pushq 0x8(%rax)`	indirect with displacement	`11111111 01110000 00001000` (3)	`ff 70 08`
`pushq 0xff(%rbp,%rcx,4)`	indirect with displacement and scaled-index	`11111111 01110100 10001101 11111111` (4)	`ff 74 8d ff`

As we discussed in class, x86-64 uses a variable-length encoding for machine instructions; some instructions are encoded as a single byte, and others require more than a dozen bytes. The `pushq` instructions to be decoded vary from 1 to 5 bytes in length. The first byte of a machine instruction contains the *opcode* which indicates the type of instruction. The opcode is followed by 0 or more additional bytes that encode additional details about the instruction such as the operands, register selector, displacement and so on. This additional bytes vary based on the particular instruction variant.

The table above shows various `pushq` assembly instructions, each with its binary-encoded machine equivalent. To interpret the binary encoding:

- The black bits identify the opcode and instruction variant. They are constant for all instructions of a given variant type.
- The red, green, and blue bits vary depending on chosen register, amount of displacement, and immediate values.
- The red bits used in register/indirect come in groups of 3. It takes 3 bits to select a register from the lower 8 registers. The selected register is encoded using the mapping:

  `%rax=000 %rcx=001 %rdx=010 %rbx=011 %rsp=100 %rbp=101 %rsi=110 %rdi=111.`

  Note in the third byte of the scaled-index variant there are two register selectors side-by-side. The left group of three bits encodes the register for the index, the right group encodes the register for the base address. (To access any of the upper 8 registers, a different instruction encoding is used that you are not responsible for disassembling)
- The blue bits encode the scale factor for the scaled-index variant. The legal values for the scale are $\{1,2,4,8\}$, thus 2 bits are required to encode a scale factor. The values are encoded with the bit patterns `00`, `01`, `10`, `11` respectively.
- The green bits encode an unsigned value of size 1-byte (displacement) or 4-byte (immediate). The 4-byte immediate is stored in little-endian order.

Your Task

You are to implement a function

```
void disassemble(const unsigned char *raw_instr)
```

The `disassemble` function takes a pointer to a sequence of raw bytes that represent a single machine instruction. How many bytes are used by the instruction is figured out during disassembling. The idea is to read the first byte(s) and use the opcode and variant to determine how many additional bytes need to be examined. You may wish to write a helper function `print_hex_bytes` to print the raw bytes, then use bitwise manipulation to extract and convert the operands and print the assembly language instruction.

For constants (immediate or displacement), use the printf format `%#x` to show hex digits prefixed with 0x and no leading zeros.

As an example, if the `disassemble` function were passed a pointer to the bytes for the last instruction in the table above, it would print this line of output:

```
ff 74 8d ff    pushq 0xff(%rbp,%rcx,4)
```

We will only test your function on well-formed instructions of the 5 `pushq` variants listed above. There is no requirement that you gracefully handle any other inputs, such as other variants of push or other instruction types or malformed inputs.

For slightly obscure reasons, there is a slight anomaly in the encoding used when `%rbp` or `%rsp` is the base register in indirect/indirect-with-displacement or `%rsp` as the index register for scaled-index. You do not need to allow for this nor make a special case for it, just disassemble as though the standard encoding is used for all registers without exceptions.

Logistics & Advice

Your implementation will go inside `disas.c`. The file `disas.c` must not contain any `main` function. You may wish to write a separate file to test your implementation.

Design/style/readability. Bitwise manipulation is known for its obscurity, so take extra care to keep the code clean and be sure to comment any dense expressions. Use macro wisely.