CS 211: Computer Architecture, Spring 2019

Programming Assignment 2: Bit Manipulation

Due: March 8, 2019 at 11:55pm.

Assignment Introduction

This assignment is designed to give you a better understanding of bits and bit manipulation. Your task is to write 3 small C bit-related programs. Your programs must follow the input-output guidelines listed in each section, with no additional or missing output.

You can assume all your input files will be in proper format as described.

Bit Introduction

Because this assignment is focused on bits, to receive credit, all parts of this assignment must only use bit operations to complete the tasks of the program. You may not use arithmetic or logic equivalents to the bit tasks being asked for.

C provides six operators for bit manipulation:

`&`	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
`|`	bitwise inclusive OR	The bits in the result are set to 1 if at least one of the corresponding bits in the two operands is 1.
`^`	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
`<<`	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand.
`>>`	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand.
`~`	complement	All 0 bits are set to 1 and 1 bits are set to 0.

For example:

```c
unsigned short x = 5, y = 2, z; //in binary 5 is 101 and 2 is 10
z = x & y; //result: 0
z = 5 | y; //result: 7
z = 5 ^ 1; //result: 4
```

An example using shifting:

```c
unsigned short x = 5;
printf("%u\n%u\n", x << 1, x >> 1);
```

First: Bit functions (35 Points)

You have to write a program that will read a number followed by a series of bit operations from a file and perform the given operations sequentially on the number. The operations are as follows:

set(x, n, v)	sets the *n*th bit of the number *x* to *v*
comp(x, n)	sets the value of the *n*th bit of *x* to its complement (1 if 0 and 0 otherwise)
get(x, n)	returns the value of the *n*th bit of the number *x*

The least significant bit (LSB) is considered to be index 0.

Input format: Your program will take the file name as input. The first line in the file provides the value of the number *x* to be manipulated. This number should be considered an unsigned short. The following lines will contain the operations to manipulate the number. To simplify parsing, the format of the operations will always be the command name followed by 2 numbers, separated by tabs. For the `set(x, n, v)` command, the value of the second input number (*v*) will always be either 0 or 1. For the `comp(x, n)` and `get(x, n)` commands the value of the second input number will always be 0 and can be ignored. Note that the changes to *x* are cumulative, rather than each instruction operating independently on the original *x*.

Output format: Your output for `comp` and `set` commands will be the resulting value of the number *x* after each operation, each on a new line. For `get` commands, the output should be the requested bit's value.

Example Execution:

For example, a sample input file "file1.txt" contains the following (except the annotation comments):

```
5               # x = 5
get   0  0      # get(x, 0), ignoring second value (0)
comp  0  0      # comp(x, 0), ignoring second value (0)
set   1  1      # set(x, 1, 1)
```

The result of the sample run is:

```
$ ./first file1.txt
1
4
6
```

Second: Bit Count functions (35 points)

In this part, you have to determine the parity of a number and the number of 1-bit pairs present in the number. Parity refers to whether a number contains an even or odd number of 1-bits. 1-bit pairs are defined by two adjacent 1's *without overlap* with other pairs.

For example:

Number	Binary sequence	Parity	Number of pairs
7	111	Odd	1
15	1111	Even	2
367	101101111	Odd	3

Input format: This program takes a single number as an argument from the command line. This number should be considered as an unsigned short.

Output format: Your program should print either "Even-Parity" if the input has an even number of 1 bits and "Odd-Parity" otherwise, followed by a tab. Your program should then print the number of 1-bit pairs present in the number followed by a new line character.

Example Execution:

```
$ ./second 12
Even-Parity   1

$ ./second 31
Odd-Parity    2
```

Third: Bit Pattern function (30 points)

In this part, you have to determine whether a number's bit representation is a palindrome. A palindrome is defined as a sequence that is the same both forwards and backwards.

We will be working with unsigned shorts which are 2 bytes or 16 bits. For example the number 384 has the binary sequence 0000000110000000 and is thus a palindrome while the sequence 0100100010111011 is not. Note that all values should be considered 16-bit, so while 5 is 101 in binary and looks like a palindrome, in 16 bits it'd be 0000000000000101, which is not.

You can and should use the same **get(x, n)** function that you created in part 1.

Input format: This program takes a single number as an argument from the command line. This number should be considered as an unsigned short.

Output format: Your program should print either "Is-Palindrome" if the input is a palindrome and "Not-Palindrome" otherwise, followed by a new line character.

Example Execution:

Some sample runs and results are:

```
$ ./third 5
Not-Palindrome

$ ./third 384
Is-Palindrome

$ ./third 1001
Not-Palindrome
```

Structure of your submission folder

All files must be included in the pa2 folder. The pa2 directory in your tar file must contain 3 subdirectories, one each for each of the parts. The name of the directories should be named first through third (in lower case). Each directory should contain a c source file, a header file (if you use it) and a Makefile. For example, the subdirectory first will contain, first.c, first.h (if you create one) and Makefile (the names are case sensitive).

```
pa2
├── first
│   ├── Makefile
│   ├── first.c
│   └── first.h (if used)
├── second
│   ├── Makefile
│   ├── second.c
│   └── second.h (if used)
└── third
    ├── Makefile
    ├── third.c
    └── third.h (if used)
```

Submission

You have to submit the assignment using Sakai. Your submission should be a tar file named `pa2.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa2`. Then, `cd` into the directory containing `pa2` (that is, `pa2`'s parent directory) and run the following command:

```
tar cvf pa2.tar pa2
```

To check that you have correctly created the tar file, you should copy it (`pa2.tar`) into an empty directory and run the following command:

```
tar xvf pa2.tar
```

This should create a directory named `pa2` in the (previously) empty directory.

The `pa2` directory in your tar file must contain 3 subdirectories, one each for each of the parts. The name of the directories should be named first through third (in lower case). Each directory should contain a C source file, a header file (if necessary) and a make file. For example, the subdirectory first will contain, first.c, first.h and Makefile (the names are case sensitive).

AutoGrader

We provide the AutoGrader to test your assignment. AutoGrader is provided as `autograder.tar`. Executing the following command will create the autograder folder.

```
tar xvf autograder.tar
```

There are two modes available for testing your assignment with the AutoGrader.

First mode

Testing when you are writing code with a `pa2` folder

(1) Lets say you have a `pa2` folder with the directory structure as described in the assignment.

(2) Copy the folder to the directory of the autograder

(3) Run the autograder with the following command

```
python auto_grader.py
```

It will run your programs and print your scores.

Second mode

This mode is to test your final submission (i.e, `pa2.tar`)

(1) Copy `pa2.tar` to the auto_grader directory

(2) Run the auto_grader with `pa2.tar` as the argument.

The command line is

```
python auto_grader.py pa2.tar
```

Scoring

The autograder will print out information about the compilation and the testing process. At the end, if your assignment is completely correct, the score will something similar to what is given below.

```
You scored
17.5  in  second
15.0  in  third
17.5  in  first
Your TOTAL SCORE =  50.0 /50
Your assignment will be graded for another 50 points with test cases not given to you
```