CS 211: Computer Architecture, Spring 2019
Programming Assignment 2: Bit Manipulation (100 points)

Instructor: Prof. Badri Nath

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 programs. Each of them will test a portion of your knowledge about bits. They are discussed below. Your program must follow the input-output guidelines listed in each section **exactly**, with no additional or missing output.

We will not give you improperly formatted files. You can assume all your input files will be in proper format as described.

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are pretty powerful. Hence, you should not look at your friend's code. See CS department's academic integrity policy at:
`https://www.cs.rutgers.edu/academic-integrity/introduction`

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 only 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.
~    one's complement        All 0 bits are set to 1 and 1 bits are set to 0.
```

```
Example in 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
```

You should try left shift and right shift yourself to understand their interactions. Sample code:

```
unsigned short x = 5;
printf("%hu\n%hu\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 nth bit of the number x to v
comp(x, n) sets the value of the nth bit of x to its complement (1 if 0 and 0 otherwise)
get(x, n) returns the value of the nth 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 to be manipulated. For the function calls this number can be considered x. This number should be considered an unsigned short. The following lines will contain the operations to manipulate the number. The format of the operations will always be the command followed by 2 numbers with tabs in between each part. For the set(x, n, v) command, the value of the second number will always be either 0 or 1. For the comp(x, n) and get(x, n) command the value of the second number will always be 0 and can be ignored.

Output format: Your output will be the resulting value of the number x after each operation, each on a new line. The get(x, n) function should only print the value of the nth bit as the value of x does not change.

Example Execution:

For example, a sample input file "file1.txt" contains:

```
5
get   0  0
comp  0  0
set   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 amount 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 2 adjacent 1's without overlap with other pairs.

For example the number 3 has the binary sequence 111 and is considered to contain 1 pair while the sequence 101101111 has 3 pairs.

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 amount 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:

Some sample runs and results are:

```
$./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.

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 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
   |-- first.c
   |-- first.h (if used)
   |-- Makefile
|- second
   |-- second.c
   |-- second.h (if used)
   |-- Makefile
|- third
   |-- third.c
   |-- third.h (if used)
   |-- Makefile
```

Submission

You have to e-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
```

Grading Guidelines

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and

source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- **You should not see or use your friend's code either partially or fully. We will run state of the art plagiarism detectors. We will report everything caught by the tool to Office of Student Conduct**.
- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. For example, programs should *not* crash with memory errors.
- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result **in up to 100% penalty**. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.

Be careful to follow all instructions. If something doesn't seem right, ask on discussion forum.