CS 211: Computer Architecture, Fall 2020
Programming Assignment 3: Understanding Data Representation

Instructor: Prof. Santosh Nagarakatte

Due: October 28, 2020 at 5pm Eastern Time.

Introduction

In this assignment, you will improve your understanding of data representation by writing C programs. Your program must follow the input-output guidelines listed in each section **exactly**, with no additional or missing output.

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are powerful. Hence, you should not look at your friend's code or use any code from the Internet or other sources such as Chegg/Freelancer. All violations will be reported to office of student conduct. See Rutgers academic integrity policy at: `http://academicintegrity.rutgers.edu/`

First: Conversion to Unsigned Binary Representation (20 Points)

In this part, your task is to write a C program that prints the unsigned binary representation of a number with a specific number of bits. The argument to the program is an input file, whose format is described in the input format. If a given number is not representable with a given number of bits (**w**), then you should print the binary representation of the number. Otherwise, you print the binary representation of the remainder when divided by 2^w.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have two positive integers separated by a space: an integer that you want to represent in binary and the number of bits to use for the representation. For each line in the input, you should print out the binary representation of the number followed by a newline character.

Example Execution:

Let's assume we have the following input file:

```
input.txt
42 12
27 3
```

Then the result will be:

```
$./first input.text
```

```
000000101010
011
```

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

Second: Conversion to Two's Complement Binary (30 points)

In this part, your task is to write a C program that prints the two's complement binary representation of a number with a specific number of bits. The argument to the program is an input file, whose format is described in the input format. If a given number is not representable with a given number of bits because the number is greater than largest positive value possible with the given number of bits, then you should print the representation for the largest positive value with the given number of bits in the two's complement representation. If a given number is not representable with a given number of bits because is smaller than smallest negative value with the given number of bits, then you should print the representation for the smallest negative with the given number of bits in the two's complement representation.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have two integers separated by a space: an integer that you want to represent in binary and the number of bits to use for the representation. For each line in the input, you should print out the binary representation of the number followed by a newline character.

Example Execution:

Let's assume we have the following input file:

```
input.txt
42 7
16 4
-9 4
```

Then the result will be:

```
$./first input.text
0101010
0111
1000
```

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

Third: Effect of Signed/Unsigned Casts (10 points)

In this part, your task is to write a C program that prints value of the number when it is cast from a unsigned number to a signed number and vice-versa. The argument to the program is an input file, whose format is described in the input format.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have two integers and two characters separated by a space: an integer that is being represented, the number of bits to use for the representation, the source representation and the destination representation. Here, `u` is unsigned representation and `s` is in signed representation. For each line in the input, you should print out the value of the number (in decimal) in the destination representation followed by a newline character.

You can assume that the unsigned value of the number is representable with the given number of bits.

Example Execution:

Let's assume we have the following input file:

```
input.txt
7 3 u s
-2 4 s u
```

Then the result will be:

```
$./first input.text
-1
14
```

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

Fourth: Decimal Fraction to Canonical Binary Fraction (30 points)

You will write a program to convert a decimal fraction to a binary fraction in the canonical representation (*i.e.*, $(-1)^s \times M \times 2^E$). For this program, M lies between $[1, 2)$. You do not have to perform any rounding for this part. You are required to print as many digits after the decimal point as specified by the input.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have a decimal fraction (use a double type to read it) and the number of bits to show in the canonical binary representation separate by space. For each line in the input, you should print the M value and E value in the canonical representation separated by space. Add a newline character after printing the output for each input.

Example Execution:

Let's assume we have the following input file:

```
input.txt
6.25 6
12.5 3
```

Then the result will be:

```
$./first input.text
1.100100 2
1.100 3
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above. Further, we will provide only positive fractions for this part of the assignment (*i.e.*, no negative numbers).

Fifth: Decimal to IEEE-FP with Rounding (40 points)

Your task is to write a program to convert a decimal fraction to IEEE-754 FP representation in a given configuration with the rounding to nearest with ties-to-even rounding mode.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have a decimal fraction (use a double type to read it), the number of the bits (**n**) in the IEEE-754 FP representation, number of bits for the exponent, and number of fraction bits. These numbers on a given line are separated by a space. For each line in the input, you should the IEEE-754 representation with n-bits followed by a new line.

Example Execution:

Let's assume we have the following input file:

```
input.txt
6.5 8 4 3
.0546875 8 4 3
.013671875 8 4 3
6.375 8 4 3
8.5 8 4 3
9.5 8 4 3
```

Then the result will be:

```
$./first input.text
01001101
00010110
00000111
01001101
01010000
01010010
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above. You can assume that input will not have NaNs and any value will not round up or down to infinities.

Sixth: Hexadecimal Bit-pattern in the IEEE-FP Format to Decimal Fraction (20 points)

Your task is to write a program that takes a hexadecimal bit-pattern and prints the decimal fractional value of the number.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have the total number of bits, the number of bits for the exponent, number of bits for the fraction, the hexadecimal bit-pattern, and the number of precision bits after the decimal point in the decimal fraction. These numbers on a given line are separated by a space. For each line in the input, you should print out the decimal fraction value with the specified number of precision bits followed by a new line.

Example Execution:

Let's assume we have the following input file:

```
input.txt
8 4 3 0x4d 2
8 4 3 0x16 7
```

Then the result will be:

```
$./first input.text
6.50
.0546875
```

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

Structure of your submission folder

All files must be included in the `pa3` folder. The `pa3` directory in your tar file must contain 6 subdirectories, one each for each of the parts. The name of the directories should be named first through sixth (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, and any additional .c or .h (if you create one) and Makefile (the names are case sensitive).

```
pa3
|- first
   |-- first.c
   |-- <additional .c .h files> (if used)
   |-- Makefile
|- second
   |-- second.c
   |-- <additional .c .h files> (if used)
   |-- Makefile
|- third
```

```
   |-- third.c
   |-- <additional .c .h files> (if used)
   |-- Makefile
|- fourth
   |-- fourth.c
   |-- <additional .c .h files> (if used)
   |-- Makefile
|- fifth
   |-- fifth.c
   |-- <additional .c .h files> (if used)
   |-- Makefile
|- sixth
   |-- sixth.c
   |-- <additional .c .h files> (if used)
   |-- Makefile
```

Submission

You have to e-submit the assignment using Canvas. Your submission should be a tar file named `pa3.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa3`. Then, `cd` into the directory containing `pa3` (that is, `pa3`'s parent directory) and run the following command:

```
tar cvf pa3.tar pa3
```

To check that you have correctly created the tar file, you should copy it (`pa3.tar`) into an empty directory and run the following command:

```
tar xvf pa3.tar
```

This should create a directory named `pa3` in the (previously) empty directory.

The `pa3` directory in your tar file must contain 6 subdirectories, one each for each of the parts. The name of the directories should be named first through ninth (in lower case). Each directory should contain a c source file, a header file 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 pa3_autograder.tar. Executing the following command will create the autograder folder.

```
$tar xvf pa3_autograder.tar
```

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

First mode

Testing when you are writing code with a `pa3` folder

(1) Lets say you have a `pa3` 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 pa3_auto_grader.py

It will run your programs and print your scores.

Second mode

This mode is to test your final submission (i.e, pa3.tar)

(1) Copy pa3.tar to the auto_grader directory

(2) Run the auto_grader with pa3.tar as the argument.

The command line is

python pa3_auto_grader.py pa3.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
7.5 in  first
10.0  in  third
10.0  in  sixth
12.5  in  second
15.0  in  fourth
20.0  in  fifth

Your TOTAL SCORE =  75.0/75.0
Your assignment will be graded for another 75 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`.
- Your compilation command with gcc should include the following flags: **-Wall -Werror -fsanitize=address -std=c11**
- 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.
- **Your folder names in the path should have not have any spaces. Autograder will not work if any of the folder names have spaces.**

Be careful to follow all instructions. If something doesn't seem right, ask on Piazza or visit during office hours.