CS 211: Computer Architecture, Summer 2018
Programming Assignment 1:
Data Representation and Computer Arithmetic

1 Introduction

This assignment is designed to help you learn the representation, interpretation, and manipulation of data in its internal representation. There are two parts. In the first part, you will implement a program `calc` to add and subtract numbers specified in different bases (multiplication is extra credit). In the second part, you will implement a program `format` that will print the decimal values of bit sequences representing integer and floating point data types.

2 Numeric Base Conversion and Calculator

Implement a program called `calc` with the following usage interface:

`calc <op> <number1> <number2> <output base>`

The first argument, <`op`>, is either the string "+", for addition, or "-", for subtraction. If you want to implement multiplication, then <`op`> can also be the string "*". (If you do implement multiplication, make sure to say so in your `readme.pdf` file so that the TAs know to check your program for this functionality.)

The next two arguments, <`number1`> and <`number2`> are *64-bit, two's-complement integers*. Each of these numbers will be given in the form of:

$$?(b|o|d|x)d_n d_{n-1}...d_1 d_0$$

which can be interpreted as: a base indicator where b means that the number is a binary number, o means octal, x means hexadecimal and d means decimal. $d_n d_{n-1}...d_1 d_0$ are the digits of the number. A decimal number may be preceded by a minus sign. Note that a minus sign is neither meaningful nor necessary for binary, octal or hexadecimal numbers as the bit pattern for these representations already covers positive and negative quantities.

The final argument, <`output base`>, gives the base for the resulting output number. Like the base indicator for the input numbers, this argument can be one of four strings: "b" for binary, "o" for octal, "d" for decimal, and "x" for hexadecimal. Your program should output the answer in the same form as the input numbers (that is, the output number should follow the regular expression given above).

Some examples:

```
$ ./calc + d1111111111111111 d1111111111111111 d
d2222222222222222
$ ./calc + b1101 b1 d
d14
$ ./calc + d999999999 d1 d
d1000000000
$ ./calc - d10 -d4 b
b1110
$ ./calc + -d10 -d4 b
b11111111111111111111111111110010
```

Given that 64-bit binary numbers can only handle numbers within a finite size and range, your program must therefore check to make sure that any input given to it will fit into a 64-bit integer. Any input that does not fit into a 64-bit integer should be flagged as an error.

Important: *You must write the base conversion code yourself.* You may not use type-casting, libraries, or output formats in `printf()`. You may use standard C arithmetic operations. You may use the C standard libraries for functionality not related to the conversion (e.g., string handling functions).

Important: If `calc` detects an error in the inputs, it should print out an error message that starts with the string "ERROR", followed by a string that gives an informative message about the error that it detected.

3 Format Interpretation

Implement a program called `format` with the following usage interface:

`format <input bit sequence> <type>`

The first argument, <`input bit sequence`>, is a sequence of 64 bits. Remember that your C program will get it as a string of 1 and 0 characters in the `argv[1]` argument to `main`. This sequence of bits represents the binary values stored in 4 contiguous bytes in memory. The leftmost bits are stored in the byte with the smallest address while the rightmost bits are stored in the byte with the largest address.

The second argument, <`type`>, gives the type that you should use to interpret the input bit sequence, and can be either `int` (integer) or `float`.

The formats for the input bit sequence is as follows. If the type is:

`int:` the format is two's complement;

`float:` the format is IEEE 754 single precision;

Note that the input bit sequence can correspond to negative numbers.

Your program should print out the decimal representation of the input bit sequence, assuming a *big endian* byte ordering. Floating point numbers should be printed in scientific notation, where a

number 1.5x10^5 would be printed as 1.5e5. For positive infinity, output `pinf`, for negative infinity, output `ninf`, and for "NaN", output `NaN`.

Here are some examples:

```
$ ./format 01000001010000100100001101000100 int
1094861636

$ ./format 10000001010000100100001101000100 int
-2126363836

$ ./format 01000001010000100100001110000100 int
1094861700

$ ./format 00000000000000000000000000000001 int
1

$ ./format 01000000110000110100001111010100 float
6.10203e0

$ ./format 00111010000111111111011000001000 float
6.102030e-4

$ ./format 10000000000000000000000000000000 float
-0.0e0

$ ./format 01000000010010010000111111011011 float
3.141593e0
```

Important: *You must write the interpretation code yourself.* You may not use type-casting, libraries, or output formats in `printf()`. You may use standard C arithmetic operations. You may use the C standard libraries for functionality not related to the value interpretation (e.g., string handling functions).

Important: If `format` detects an error in the inputs, it should print out an error message that starts with the string "ERROR", followed by a string that gives an informative message about the error that it detected. For this program, you can assume that any input bit sequence that is shorter or longer than 32 bits is erroneous.

4 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa1.tar` that can be extracted using the command:

```
tar xvf pa1.tar
```

Your tar file must contain:

- `readme.pdf`: this file should describe your design and implementation of the `calc` program. In particular, it should detail your design, any design/implementation challenges that you ran into, and an analysis (e.g., big-O analysis) of the space and time performance of your program.
- `makefile`: there should be at least the following rules in your `makefile`:

  `all` build both your `calc` and `format` executables.

  `calc` build your `calc` executable.

  `format` build your `format` executable.

  `clean` remove all object files and executables. Prepare for rebuilding from scratch.

- source code: All source code and header files necessary for building your `calc` and `format` executables. These should at least include calc.c and format.c.

You can build your pa1.tar file in the following steps:

1. Put all the files you want to hand in in a subdirectory called pa1.
2. In the parent directory that contains pa1, invoke tar:

```
tar cvf pa1.tar pa1
```

The arguments to tar are cvf. The c tells tar to create a new archive file. The f tells tar that the next command line argument is the name of the output file. The v just makes tar list the files it's putting into the archive.

We will compile and test your program on the iLab machines so you should make sure that your code can be extracted from your pa1.tar file on the iLab machines and that your program compiles and runs correctly on these machines. You must compile all C code using the gcc compiler with the `-Wall` flags.

5 Grading Guidelines

5.1 Functionality

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 a 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 make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. *In particular, your code should be adept at handling exceptional cases.*

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

5.2 Design

Having said the above about functionality, design is a critical part of any programming exercise. In particular, we expect you to write reasonably efficient code based on reasonably performing algorithms and data structures. More importantly, you need to understand the performance (time & space) implications of the algorithms and data structures you chose to use. Thus, the explanation of your design and analyses in the `readme.pdf` will comprise a non-trivial part of your grade. *Give careful thoughts to your writing of this file, rather than writing whatever comes to your mind in the last few minutes before the assignment is due.*

5.3 Coding Style

Finally, it is important that you write "good" code. Unfortunately, we won't be able to look at your code as closely as we would like to give you good feedback. Nevertheless, *a part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.
- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `i`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.
- Error and warning messages should be printed to `stderr` using fprintf().