Assignment 2: Signed Number Systems

Objectives

Write C programs that perform the followings:

1. Converts any **signed integer** number into a **sign magnitude binary** number.
2. Converts any **sign magnitude binary** number into a **signed integer** number.
3. Converts any **signed integer** number into a **tows complement binary** number.
4. Converts any **tows complement binary** number into a **signed integer** number.
5. Converts any **signed integer** number into a **biased-127 binary** number.
6. Converts any **biased-127 binary** number into a **signed integer** number.
7. Converts any **signed real** number into a **floating point binary** number.
8. Converts any **floating point binary** number into a **signed real** number.

Specifications

In the real world we often use signed integer and signed real numbers. Computers often store signed integer numbers either as sign magnitude binary numbers or twos complement binary numbers. Computers occasionally store a small and fixed range of signed integer numbers as biased-127 binary numbers. Most of the modern computers store signed real numbers as floating point binary numbers.

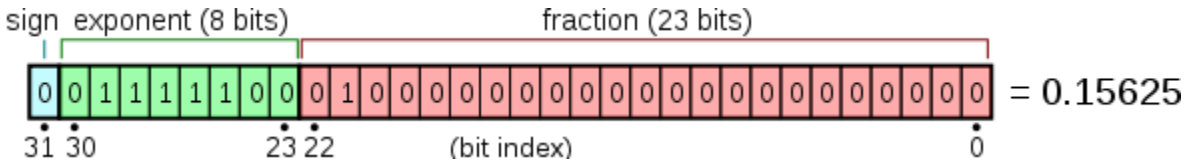
The sign-magnitude binary format is the simplest conceptual format. To represent a number in sign-magnitude, computers simply use the leftmost bit to represent the sign, where 0 means positive, and the remaining bits to represent the magnitude (absolute value).

A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two. The most significant bit of a twos complement number also represents the sign of the number.

Biased-127 notation stores a number N as an unsigned value N+127, where 127 is the bias. 127 is typically half the unsigned range.

Floating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of a sign bit, an exponent, and a significand. Significand holds a fixed number of binary digits and the length of the significand determines the precision to which the numbers can

be represented using the corresponding floating point number system. Binary radix point position is assumed always to be somewhere within the significand-often just after the most significant digit. The sign bit represents whether the significand is positive or negative. Logically exponent is a signed integer that represents the scale of the number by which its magnitude is going to be multiplied. To derive the value of the floating-point number, the significand is multiplied by the base raised to the power of the exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent—to the right if the exponent is positive or to the left if the exponent is negative. Therefore, a big positive exponent will evaluate the number into a very big number and a big negative exponent will evaluate the number into a very small number. In this assignment we are focusing on IEEE 754 Single Precision Floating Point Number System that uses a 32 bit word to represent the floating point numbers. IEEE 754 significand is 23 bits long and it most significant bit is always 1 (normalized, i.e., no leading zeros) and the binary radix point is always assumed after the most significand bit. IEEE 754 exponent is 8 bit long and biased-127 binary representation is used to represent the singed integer exponent. An example of a layout for 32-bit floating point is:



Tasks

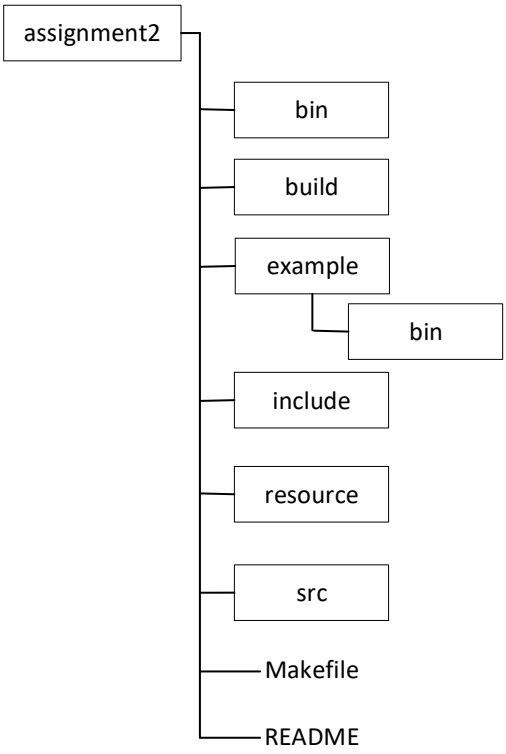
1. You will submit this assignment using **GIT submission system**. A central repository named '**assignment2**" has been created for this assignment.
2. Create your own fork of **assignment2** on the central GIT repository using following command.

   ssh csci fork csci261/assignment2 csci261/$USER/assignment2

3. Go into your **csci261** folder that you have created in your home folder for A*ssignment1* and create a clone of your forked **assignment2** repository using following command.

   git clone csci:csci261/$USER/assignment2

4. Repository **assignment2** has been organized as follows:



A **README** file template has been placed in the root of the application development folder. The README file gives a general idea of the application, technologies used in developing the application, how to build and install the application, how to use the application, list of contributors to the application, and what type of license is given to the users of the application. You will need to complete the **README** file before your final submission.

The **specifications** or **header** files (**signed_convert.h** and **utility.h**) have been placed in **include** sub folder.

All **source codes** (**signed_convert.c**, **utility.c**, and **main.c)** go into **src** sub folder**.** You need to implement the functions defined in the header file **utility.h** in your source code file **utility.c**. You need to implement the functions defined in the header file **signed_convert.h** in your source code file **signed_convert.c**. The **main.c** file in the **src** folder is complete, it uses the functions from your source code in **signed_convert.c** and **utility.c.** You don't need to must not modify **main.c** file.

All **object** files will be placed in **build** sub folder and **executable** files in **bin** sub folder by **make**.

Data files are usually placed in **resource** folder. In this assignment there is no data file in resource folder.

A **Makefile** has been placed in the root of the application development folder. The **Makefile** does the followings and you don't need to and must not modify **Makefile**:

a) Defines and uses **macros** for each **GCC flag** that you are going to use to compile the codes and objects.
b) Defines and uses **macros** for each **sub folder** of the application, e.g., **src**, **include**, **resource**, **build**, and **bin**.
c) Uses GCC **debug flag** to facilitate debugging using **gdb**.
d) Uses GCC **include flag** to specify the path of application's custom header files so that the code does not need to specify relative path of these header files in **#include** pre-processor macro.
e) Creates individual object file (***.o**) into **build** folder from each source (***.c)** file of **src** folder.
f) Links the necessary object files from the **build** folder into a single executable file in **bin** folder.
g) Runs the main executable of the application from **bin** folder.
h) Cleans or removes files from both **build** and **bin** folders using **PHONY target** named **clean**.

5. One example executable (**signed_convert**) of this application has been placed in the **example/bin** folder. You can run this executable to get an idea what is expected from you in this assignment. To run the example executable type followings:

   make run-example

   This example executable has been built and tested in Linux Debian machines available in the labs. Run this executable in other kind of machines at your own risks. Make clean command will not delete this example executable and you should not delete it either.

6. Type following at the command prompt to clean previously built artefacts (binary and object files) of the application:

   make clean

7. Type following at the command prompt to build the application from your own source code:

   make

8. Type followings at the command prompt to run your own application:

   make run

9. Make sure you can compile, link, run, and test this application error and warning free.

10. Complete the **README** file. Therefore, you need to give the general description of the application, technologies that are used in the application, how a user can build (compile and link) and install the application, how a user can run the application after the installation. Mention instructor's name and your name in the list of contributors. Give GPL license to the users to use the application. You can google to find README examples if you are not sure how to write one.

11. Organize and comment your code to make it easy to understand. Make sure you have typed **your name** and **student number** in the top comment section in each **.c** file. Make sure you have deleted all debugging print codes that you were using to debug your code during your development time but not necessary in the final code. Also, make sure you have deleted all commented out codes from your final submission.

12. Continue your work in your cloned or local **assignment2** repository and **commit** and **push** your work to your central **assignment2** repository as it progresses.

Deadline and Submission

The **deadline** to **demonstrate** this assignment **in the lab** is **February 07, 2022** for lab section **S22N01** and **February 08, 2022** for lab sections **S22N03** and **S22N04.** The assignment will be evaluated to zero without lab demonstration.

The **deadline** to **submit the code** of this assignment is **11:00 PM** on **February 08, 2022** for all sections**.** The assignment will be evaluated to zero without code submission.

Commit and **push** your work from your local repository to your remote repository regularly. Use following git commands to commit and push your work from your local repository to your remote repository from your project's root folder:

git add --all

git commit –am"Commit message"

git push origin master

Remember '**git add --all**' has double dashes before 'all'. You can also use '**git add .**' dot option instead of 'all' option. Both options do the same, add all the new and the modified files into the commit index. If you want to add only a specific file into the commit index, you can specify the relative path of the file instead of dot or 'all' option. For example, if you want to add only **prog.cpp** file of the src folder into the commit index, you can use '**git add src/prog.cpp**' command. You can also include multiple files or paths separated by space in the same '**git add**' command instead of using multiple 'git add' commands. Command '**git add**' is necessary before each '**git commit**'

command. If you skip 'git add' command before a 'git commit' command, it does not perform the actual commit of the new and modified files into the repository. Always type a meaningful message in your '**git commit**' command's **'-m**' option. For example, if you are committing after adding all the necessary comments into your 'Makefile', use '**git commit –m"Added Makefile comments**"'. Remember there is a single dash before m in 'git commit' command. It is not recommended to make a huge commit instead of a number of small commits. It is also recommended to avoid unnecessary commits. Commits should reflect the check points of your software development milestones. Command 'git push' is necessary to push the local commit to the remote repository. If you skip '**git push**' after a '**git commit'**, your local and remote repository will become unsynchronized. You must keep your local and remote repositories synchronized with each other using '**git push**' after each '**git commit'**.

You will find most useful git commands in this **git cheat sheet** from GitLab. You will be allowed to commit and push until the deadline is over. Incremental and frequent commits and pushes are highly expected and recommended in this assignment.

Evaluation

Module	Functions	Marks
utility.c	*reverse*	**1**
	add_sign	**2**
	extend_integer_binary_to_word_size	**2**
	extend_fraction_binary_to_word_size	**3**
	are_binary_digits	**1**
	are_decimal_digits	**1**
	is_binary	**1**
	is_decimal	**1**
	is_biased_127	**2**
	is_biased_127_binary	**2**
	Is_significand	**2**
	is_real	**1**
	is_floating_point	**3**
	get_integer_part	**3**
	get_fraction_part	**2**
convert.c	*integer_to_binary*	**1**
	fraction_to_binary	**3**
	binary_to_int	**2**
	one_bit_add	**3**
	get_magnitude_binary	**2**
	to_sign_magnitude	**3**

	from_sign_magnitude	**3**
	ones_complement	**2**
	twos_complement	**2**
	to_twos_complement	**3**
	from_twos_complement	**4**
	int_to_biased_127	**2**
	to_biased_127	**2**
	int_from_biased_127	**2**
	from_biased_127	**2**
	normalize	**4**
	to_floating_point	**5**
	get_significand_value	**3**
	from_floating_point	**5**
README		**05**
Code Quality and Comments		**15**
Total		**100**