Assignment 3: Integer Arithmetic

Objectives

Write C programs that perform the followings:

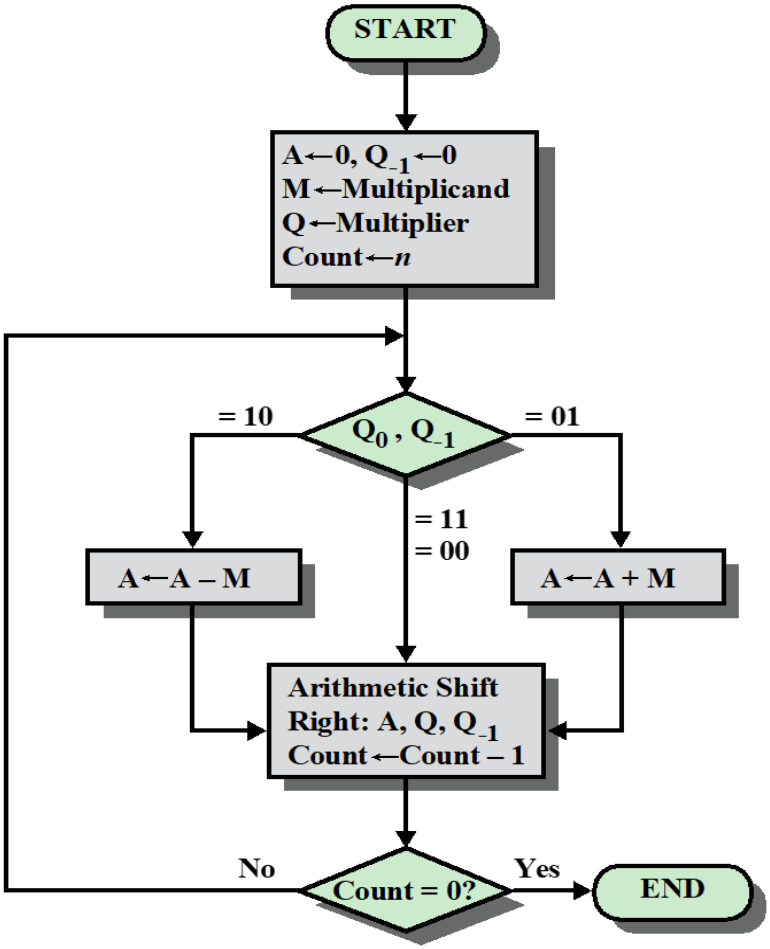
1. Adds a **twos complement binary** number to another **twos complement binary** number.
2. Subtracts a **twos complement binary** number from another **twos complement binary** number.
3. Multiplies a **twos complement binary** number by another **twos complement binary** number.
4. Divides a **twos complement binary** number by another **twos complement binary** number.
5. Adds a **signed integer** number with another **signed integer** number.
6. Subtracts a **signed integer** number from another **signed integer** number.
7. Multiplies a **signed integer** number by another **signed integer** number.
8. Divides a **signed integer** number by another **signed integer** number.

Specifications

Add operation on two twos complement numbers are very simple, bitwise addition of two numbers always yields the expected result, irrespective of their sign bit values. If two numbers are added, and both are either positive or negative, then overflow occurs if and only if the result has the opposite sign of the numbers that have been added.

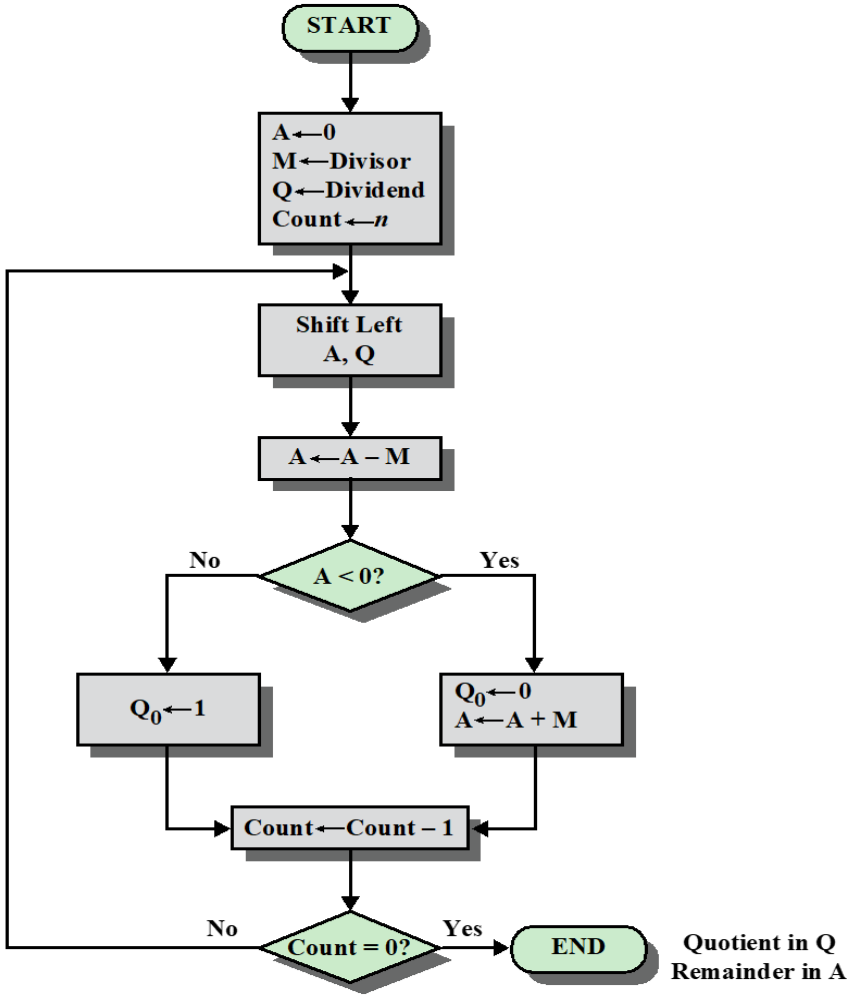
To subtract a twos complement number (subtrahend) from another twos complement number (minuend), we need to take the 2's complement (negation) of the subtrahend and add it to the minuend.

We use Booth's Multiplication Algorithm to multiply one twos complement number (multiplicand) by another twos complement number (multiplier). Following is the flowchart of Booth's Multiplication Algorithm. The flowchart assumes **A**, **Q**, and **M** are 32-bit registers and $\mathbf{Q}_{-1}$ is a 1-bit register. Multiplicand and Multiplier are loaded into **M** and **Q** respectively. Both **A** and $\mathbf{Q}_{-1}$ are initialized to **zero** to start with**.** At the completion of the algorithm the result of the multiplication operation is available in **A** and **Q** registers. Register **A** holds the most significant 32 bit and **Q** holds the least significant 32 bit of the result.



One way to do twos complement division is to convert the operands into unsigned values and perform unsigned division operation, at the end, to account for the signs by complementation where needed.

Following is the flowchart of Unsigned Division Algorithm. Like Booth's multiplication algorithm, this division algorithm assumes **A**, **Q**, and **M** are 32-bit registers. Divisor (D) and Dividend (V) are loaded into **M** and **Q** respectively. Register **A** is initialized to **zero** to start with**.** At the completion of the algorithm the quotient (Q) of the division operation is available **Q** register and the remainder (R) is available in register **A.**



The magnitudes of Quotient (Q) and Remainder (R) are unaffected by the input signs and the signs of Q and R are easily derivable from the signs of Divisor (D) and Dividend (V).

sign(R) = sign(D)

sign(Q) = sign(D) * sign(V)

Tasks

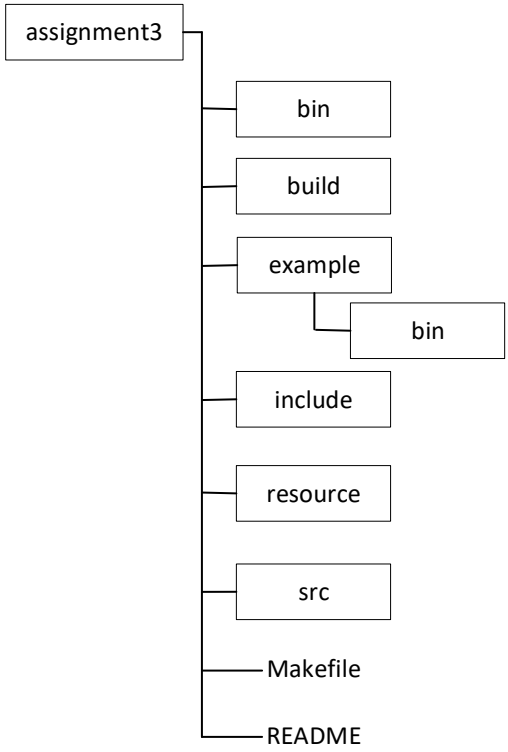
1. You will submit this assignment using **GIT submission system**. A central repository named '**assignment3**" has been created for this assignment.
2. Create your own fork of **assignment3** on the central GIT repository using following command.

ssh csci fork csci261/assignment3 csci261/$USER/assignment3

3. Go into your **csci261** folder that you have created in your home folder for your *Assignment1* and create a clone of your forked **assignment3** repository using following command.

   git clone csci:csci261/$USER/assignment3

4. Repository **assignment3** 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 (**convert.h, integer_arithmetics.h,** and **utility.h**) have been placed in **include** sub folder.

All **source codes** (**convert.c**, **integer_arithmetics.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

convert.h in your source code file **convert.c**. You need to implement the functions defined in the header file **integer_arithmetics.h** in your source code file **integer_arithmetics.c**. The **main.c** file in the **src** folder is complete, it uses the functions from your source code in **convert.c, integer_arithmetics.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. An example executable (**integer_arithmetics**) 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 this example executable type followings:

   make run-example

   These example executables have been built and tested in Linux Debian machines available in the labs. Run these executables 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 executable (**integer_arithmetics**):
   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 **assignment3** repository and **commit** and **push** your work to your central **assignment3** repository as it progresses.

Deadline and Submission

The **deadline** to **demonstrate** this assignment **in the lab** is **March 07, 2022** for lab section **S22N01** and **March 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 **March 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**
	count_leading_ones	**2**
	group_arithmetic_shift_right	**4**
	group_logical_shift_left	**4**

	are_binary_digits	**1**
	are_decimal_digits	**1**
	is_binary	**1**
	is_decimal	**1**
convert.c	*integer_to_binary*	**1**
	get_magnitude_binary	**2**
	binary_to_int	**2**
	one_bit_add	**3**
	ones_complement	**2**
	twos_complement	**2**
	to_twos_complement	**3**
	from_twos_complement	**4**
integer_arithmetics	*copy_twos_complement*	**3**
	add_twos_complement	**5**
	subtract_twos_complement	**5**
	multiply_twos_complement	**7**
	divide_twos_complement	**7**
	add_integer	**5**
	subtract_integer	**5**
	multiply_integer	**5**
	divide_integer	**5**
README		**05**
Code Quality and Comments		**10**
Total		**100**