# CMPT 295 Assignment 1 (2%)

Submit your solutions by Friday, January 18, 2019 10am.
(*Reminder:* Late assignments will not be marked.)

0. [0 marks] *Care-Package Download*

   Download the care package from the course web page and extract all files to `sfuhome`.

   *Note:* There will be one care package per assignment.

1. [3 marks] *Web Submission Test*

   Open the web browser and access the secure URL:

   `https://courses.cs.sfu.ca/`

   This is where you will receive grade updates, and where you will submit your electronic work. For this question, you will submit the file `a0.zip` from the care package to the activity "Assignment 0".

   - Read the documentation on this page very carefully, and then bookmark the page.
   - Follow the instructions and submit `a0.zip` for Assignment 0.

2. [3 marks] *Compilers and Interpreters*

   The numerical encoding/decoding of instructions in low-level *machine language* is difficult at best for human programmers to code with accuracy. The technique of using strings of 0s and 1s (or hex) to represent instructions obfuscates the meaning of the code, and is easily prone to errors in the translation. Though programming in assembly language mitigates these two issues to some degree, usually programmers write their instructions in a high-level language so that their algorithms are expressed in a more "human readable" representation. To translate a program written in a high-level language into an equivalent low-level machine language, either a *compiler* or an *interpreter* is used.

   A compiler translates the entire program from the source language into machine language to form an *executable*, which can then be run directly on the machine itself. This translation happens once, i.e., the compiler is no longer involved when the executable is run. `gcc` is one example of a compiler.

   An interpreter, on the other hand, translates instructions by executing them or simulating them on the machine, one step at a time. The interpreter itself runs while the program is running, and because of this extra overhead, interpreted code usually runs slower than compiled code. The Python interpreter and the Java Virtual Machine are two examples of interpreters.

   But the differences between compiled code and interpreted code are slight: there are programs whose purpose it is to interpret compiled programs in a sort of computerized virtual reality. For instance, suppose you wished to play the original Impossible Mission — circa 1984 — for the Commodore 64. You have the compiled program, but sadly, you don't have a Commodore 64 machine to run it on. That's okay, thanks to interpreters. You can install a program that can interpret Commodore 64 instructions and run Impossible Mission on your Intel machine, displaying the result as a windowed application.

   Such an interpreter is a special type of virtual machine. What is it called?

3. [7 marks] *Simulation*

   - Changing directories to `care-a1/q3`, you will find two source files and a makefile.
   - Build the executable `x` using `make`, and run it.
   - The mainline routine calls a subroutine `mystery(char *str, int n)` where the variable `str` is an array of `char`, i.e., an ASCII string. By modifying the mainline routine and hand tracing the assembly code, answer the following questions.

*Variable map:*

- `%rdi` is the first argument, i.e., `char *str`;
- `%esi` is the second argument, i.e., `int n`;
- `%eax` is the return value, an `int`.

(a) [2 marks] What is the output? What if you change the value of `n` to be `14`? Re-run the program for `n` = `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`, `14` and `15`, and tabulate your results.

(b) [2 marks] Using a single plain-English sentence, describe the function of the subroutine `mystery`.

(c) [1.5 marks] Describe the purpose of the value maintained in register `%rcx`.

(d) [1.5 marks] Does the main loop progress forward or backward through `str`? Justify your answer.

4. [7 marks] *Integer Multiplication*

Some (older) machines were restricted in their arithmetic operations. Addition and subtraction were available, but multiplication and division were not, and therefore they had to be written as *subroutines* by the programmer. Your goal is to write an assembly routine that performs unsigned integer multiplication, *without* using the `imul` instruction (or `mul`, or any instruction that multiplies for you).

Unless you are pursuing the bonus (see below), your algorithm should follow the algorithm of adding `a` to itself `b` times (or vice versa). That is, $\texttt{a} \cdot \texttt{b} = \underbrace{\texttt{a} + \texttt{a} + \cdots + \texttt{a}}_{\texttt{b times}}$.

*Getting Started:*

- Change directories to `care-a1/q4`. There you will see the base code to multiply two unsigned numbers `a` and `b`.
- You will modify the subroutine in `mul.s`, replacing the two instructions that reside there with your solution.

*The Specification:*

- The register `%edi` will contain the argument `a`.
- The register `%esi` will contain the argument `b`.
- The register `%eax` will carry the return value, the product of `a` and `b`.
- You may use registers `%rax`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10` and `%r11` as *scratch registers*, i.e., you may modify their values without penalty. Plan your register usage accordingly!
- You may not modify the values of registers `%rbx`, `%rbp`, `%rsp`, `%r12`, `%r13`, `%r14` and `%r15`.
- You may not modify the values of any external memory locations.

*You will submit:*

(a) [5 marks] an electronic copy of your `mul.s` assembly source, submitted to the `courses` server. This code will be tested for correctness using the mainline routine, but with different inputs. Please test your code accordingly.

(b) [2 marks] a hard copy of your `mul.s` assembly source. Your source should be well documented with *high-level comments*, in such a way that any other programmer could read your code and understand it. Also, your documentation shall include a synopsis of the algorithm you used to perform the multiplication.

(c) [3 BONUS marks] The quick multiplication of integers is a subject of great importance to both practical and theoretical computer scientists. For large integers, it is possible to perform the integer multiplication $m \cdot n$ using many many less than $\min\{m, n\}$ `add` instructions. Can *you* do better in *your* code?

To attempt the bonus, submit your faster algorithm in `mul-bonus.s`.