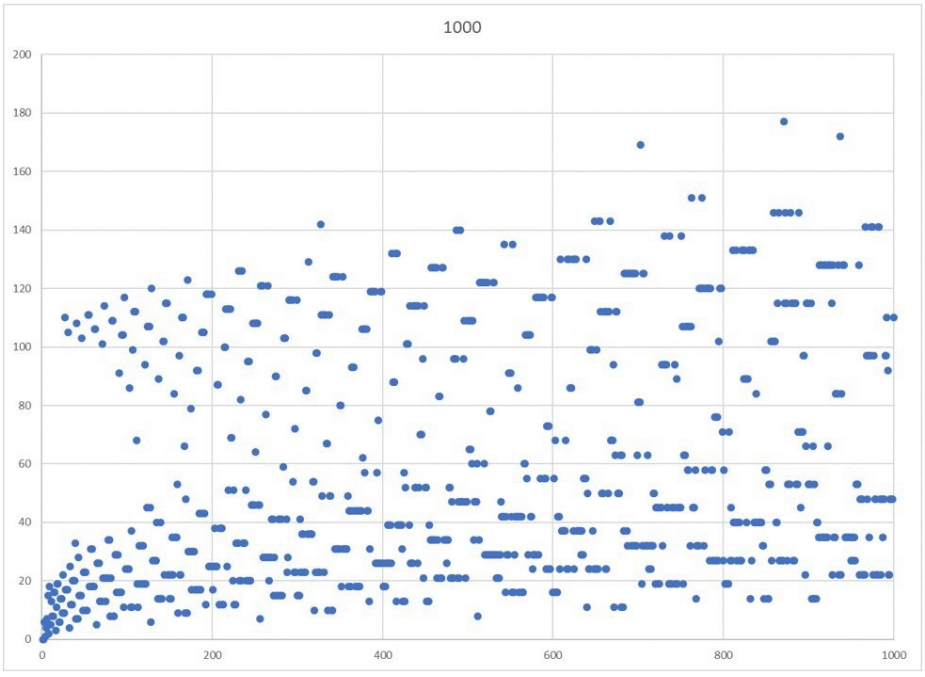
CS2240 Spring 2023
Programming Project #1 – The Collatz Conjecture
Due: Friday, March 10, 2023, 11:59pm

A proof of the so-called “Collatz Conjecture” has been an unsolved problem in mathematics, and a monetary prize for a proof has gone unclaimed for a long time. The conjecture starts with some arbitrary starting number, which may be any integer greater than 1. The goal of the “game” is to reduce the starting number to the number 1 in a finite number of steps, where each step will make the number either increase or decrease to a new current number. At each step there are two possible operations that may occur: If the current number is even, divide it by 2. If the current number is odd, multiply by 3 and add 1. Thus the initial starting number could be increased or decreased, in some cases by large amounts, in successive steps, but the sequence stops when the decreasing steps eventually overcome the increases, and the number gets reduced to 1. The unproven conjecture originally proposed by Collatz is that this prescribed sequence will always terminate at 1 in a finite number of steps no matter what starting integer is chosen.

We will study this problem numerically for this programming project. That is, we will trace the number of steps required in a sequence to reduce some starting number to 1, with a different sequence run for each different starting number ranging from 2 to several hundred. You can read about the details of this problem at https://en.wikipedia.org/wiki/Collatz_conjecture. The programming task here is to produce a graphical plot on the Discovery Board screen with a pattern similar to the plot found on the Wikipedia page:

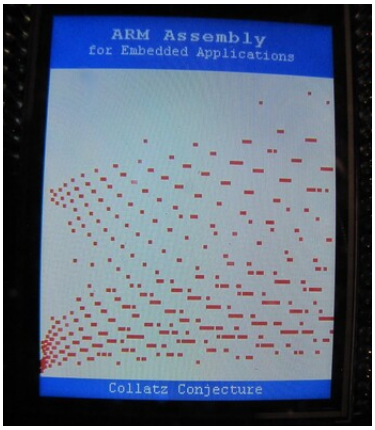


The horizontal axis of the plot is the starting number for each sequence, and the vertical axis is the number of steps required to reduce that starting number to 1. As you can see, some starting numbers require a large number of steps to eventually get to 1, but the finite limit of the vertical axis over all attempted starting numbers supports the conjecture. As detailed in the Wikipedia article, this problem has been studied numerically over a very wide range of starting integers, and a number that does not reduce to 1 with a finite number of the prescribed steps has never been found, but of course this is not a

formal mathematical proof of the conjecture. (Depending on interest, you may find the video at https://www.youtube.com/watch?v=094y1Z2wpJg to also be a good background.)

Download the C code file Project1- CollatzMain.c and the template assembly source file Project1-Collatz.s. Your project code will resemble the lab exercises with two source code modules, a main program in C, which in the interest of time is provided for you, and an assembly module that you will code. The division of computational tasks between the main C program and the assembly code functions is typical in high-performance applications, where the (typically slower) graphical interface is left to the higher level code, but a high-speed computational task is coded in assembly code.

Note in the main C program the function '`collatz()`' is declared to expect one 32-bit unsigned integer argument, and return an unsigned integer value. This function you will code in the assembly language module to take the function argument as the starting integer, run the sequence determined by that initial value, accumulate a count of the number of steps needed to reduce the number to 1, and then calculate and return a **screen row coordinate** that corresponds to the graphical plot. For a sequence that required a count of *m* steps from the starting number to reach 1, the row coordinate your function returns must be calculated with the expression $r = 300-((m*4)/3)$. This will map a count of 0 (not actually possible for starting numbers greater than 1) to the row coordinate 300, for a plot point near the bottom of the screen, and a count of 180 to row 60, for a plot point just below the top margin of the screen. The main C program will iteratively call your assembly function with increasing argument numbers, and draw a cluster of nine pixels around each plot point. The column pixel coordinate is incremented every four trials in the C code to correspond to the starting number, and the row coordinate for the plot is that returned by your function. This should produce the plot shown in the photo below, with the expected pattern.



Note that a critical requirement for the Collatz algorithm is to decide at each step whether the current integer number is even or odd. From the basic properties of binary numbers, this is particularly easy, as the least significant bit b_0 will be 0 for an even number and 1 for an odd number. So you will need to make the execution for a few statements in your code conditional on the state of the b_0 bit of some register that holds the current number. As you have seen in the lecture slides, there are some condition codes that test the "C" carry bit in the condition flags, some codes that test the "Z" zero bit, and other codes that test the "N" sign bit. How do we arrange for the b_0 bit of a register to affect these flags and be tested by conditional instructions? Consider these two instructions, and how either of them may be helpful in your programming task:

```
LSLS Rn, #31
```

or

```
RORS Rn, #1
```

where `Rn` is some register you are using to hold the current integer number.

Use a conditional branch instruction to form the main loop in your function that iterates the steps and terminates when the argument number is reduced to 1. Once you determine at each step whether the current number is even or odd, you may use a conditional branch to select which type of operation, increasing or decreasing, is applied, but using an If-Then block for this task would be good ARM

programming form. Until we study the push-pop stack and gain the availability of more registers, you must only change the values of registers R0, R1, R2, R3, or R12.

Run your code and verify that it produces a display similar to that shown above. Upload your code and a photo of your screen to Canvas. Remember the helpful ‘SVC #0’ debugging instruction available with your Discover Board library! That can be inserted anywhere in your assembly code any time you need to clarify exactly what is in the registers. Hint: My solution uses about 20 lines of assembly code in the `collatz()` function, and only uses R0 through R3. Make sure, however, that any debugging breakpoints you inserted for debugging your code have been removed in the version you submit.