Floating Point Programming

As a vehicle to practice using floating point instruction, this program will throw darts at a square with sides of two units. It turns out the fraction of darts that fall within the unit circle is... wait for it... PI.

The idea is to choose two random numbers from zero to one. Treat these as an x and y coordinate. Calculate the distance to the origin. If the distance puts the point inside the unit circle, record the trial as a hit.

Finally, divide the number of hits by the number of trials and multiply by four. The result will be an approximation of PI using stochastic methods.

Note this means we are throwing darts at one quadrant of a square with sides of 2 units, not the whole square. The difference means we must multiply by 4 at the end.

Number of trials to run

This will come from the command line as in:

```
~/pi $ ./a.out 100000
Executing: 100000 iterations.
Hits: 78443
Approximation: 3.137720
~/pi $
```

If the command line argument is *not* given, use a default of 100000.

Remember that all AARCH64 instructions are 32 bits long. An implication of this is you can't simply do:

```
    mov         x0, 1000000
```

Since the constant cannot fit along with op codes into four bytes. A way to get around this that comes readily to mind is to put the constant in RAM and `ldr` it into a register.

Vetting the command line argument

You are not responsible for doing this. Assume that all command line arguments are valid.

Converting command line argument to integer

`atoi`

Seeding the random number generator

In your previous C and C++ programs, you will have done (code snippets follow):

```
#include <ctime>

srand((unsigned int) time(nullptr));
```

You must seed the RNG in this way. But in assembly language.

Getting a random number in the right range

`rand()` returns an integer between 0 and `RAND_MAX`. In C and C++ you would do:

```
// Produces result between 0 and 1.
float v = float(rand()) / float(RAND_MAX);
```

You must do this in your program. What value is RAND_MAX? Write a tiny C++ program on the ARM and print out the value. Or, put `RANDMAX` into an IDE and ask the IDE to locate its definition.

You must write a subroutine (function) which returns a random number in the right range. Call it `randf` so I can find it easily.

Converting integers to doubles

In the above, the integers were converted to floats with a cast. In assembly language you must code the instructions which perform the cast yourself. Look up `scvtf` which stands for "signed convert to float".

FP registers

Just like `x0` through `x30`, there exists `d0` through `d30`. Among the float registers, `d29` and `d30` are not special. However, non-scratch float registers must still be saved and restored in functions.

FP ops

You'll use instructions like `fmul`, `fdiv`, `fsqrt`, `fmov` and `fcmp`.

Aliases For Registers

We have previously advised you to create a "bible" documenting which registers are used for what. Here is another way of aiding you to:

- better understand your code, and
- avoid using the wrong register in an instruction.

The idea is to give specific registers aliases, symbolic names that mean something to your code rather than just a letter and number. Here is an example:

```
LOOP_MX .req        x19
LOOP_CT .req        x20
HITS    .req        x21

RND_MF  .req        d20
DTMP    .req        d21
```

Printing FP

`printf` will be your friend. Like always, the format string address goes in `x0`. A `%f` found in the format string tells `printf` to look in the FP registers starting with `d0` as the first value.

Added Challenge

The program as specified above will print its result only at the end of its computation. If you specify a very large number of loops, this can take a long time. The shell prompt will just sit and you won't know if your program is working or not.

As an added challenge, add functionality that tests the loop counter and from time to time, such as every 2048 loops, prints the results so far.

Note that this can produce a lot of lines of output. To deal with this, create a single line format string (for `printf()`) that also prints some *old school* cursor control sequences.

Add this to your single line intermediate results:

```
\033[1;1H\033[2J
```

These characters will be interpreted by the terminal (console) as meaning:

- Move the cursor to line 1 column 1, and
- Erase everything below this position.

`\033` is the ESCAPE character represented by its value in *octal*. *Octal* is indicated by the leading bash (back slash) and the leading 0.