CS-473:
System programming
for
Systems on Chip

Practical work 2

The Mandelbrot set

Version:
1.0

Contents

1 Introduction **1**
1.1 Prerequisites . . . 1
1.2 Number systems . . . 1

2 Exercises **2**
2.1 Prerequisites . . . 2
2.2 Fixed point Mandelbrot . . . 2
2.3 Your own floating point format . . . 3
2.4 Grading . . . 3
2.5 Workflow . . . 3
2.6 Not a bug, or maybe yes? . . . 3

1 Introduction

1.1 Prerequisites

On moodle you find the file `mandelbrot.zip`; it contains the complete source code of the floating-point version of the Mandelbrot set. Download this file, and unzip it in the directory `programms`, where you also have the `helloWorld` and `bouncingBall` programs of last week.

1.2 Number systems

Reminder: The float data type requires 32 bits of memory. It contains the sign in the MSB, followed by 8 bits, which describe the exponent of the number. Subsequently, there are 23 bits which correspond to the fraction of the value 1. This means that a given number is multiplied or divided by 2 as often as it fits into the range between 1 and 2. The number of multiplications or divisions corresponds to the exponent (8 bits). Then 1 is subtracted from the intermediate result, so that the number gets the value between 0 and 1. This is stored as a fraction in the remaining 23 bits.

As you can imagine from the description, doing floating point calculations on a μP is not evident. Also the hardware implementation of a *floating point unit (FPU)* is very "big". This is the reason why most embedded systems do not have a *FPU*. This results into a software implementation of the floating-point arithmetic's in form of a library.

On the other hand, embedded processors handle efficiently integers. Although integers do not have fractions, we can easily "transform" them to fixed-point numbers by putting "somewhere" the decimal point. Note that the compiler does not has a notion of your fixed-point format. Hence it is up to you to "handle" the position of the point. We often talk here about the `Qx.y`-format where `x` is the number of bits before the decimal point, and `y` the number of fractional bits. Note: $x + y = 32$ in case of a 32-bit based integer. Example:

```
typedef int32_t fxpt_8_24; // !< Q8.24 fixed-point type
```

Furthermore, it is also possible to define our own floating-point format (that is more appropriate for the given algorithm). By not using the floating-point library, but using our own, also can improve execution speed. In-lining these *functions* also helps.

2 Exercises

2.1 Prerequisites

Familiarize yourself with the floating-point version of the Mandelbrot set. Compile the program by:

```
cd mandelbrot/sandbox
../compile_flpt.sh
```

Make sure that you do not have any errors. Then, download the `fractal_flpt.mem` file to your virtual prototype and see the result.

2.2 Fixed point Mandelbrot

We are first going to transform the floating-point version of the Mandelbrot set into a fixed-point version. To be able to do this, you can proceed according to following steps:

1. Think of a proper `Qx.y`-format for this algorithm. Note: although we are not going to zoom into the Mandelbrot set in this exercise, there will be an exercise where this functionality is added.
2. Create the file `fractal_fxpt.h` where you define this format with a typedef, and you modify:

   ```c
   //! \brief Pointer to fractal point calculation function
   typedef uint16_t (*calc_frac_point_p)(float cx, float cy, uint16_t n_max);

   uint16_t calc_mandelbrot_point_soft(float cx, float cy, uint16_t n_max);

   void draw_fractal(rgb565 *fbuf, int width, int height,
                     calc_frac_point_p cfp_p, iter_to_colour_p i2c_p,
                     float cx_0, float cy_0, float delta, uint16_t n_max);
   ```

   to account for your new type.
3. Copy the file `fractal_flpt.c` to the file `fractal_fxpt.c`. Modify the required functions in `fractal_fxpt.c` to perform the fixed-point Mandelbrot algorithm. Following functions can be left untouched:

   ```c
   rgb565 iter_to_bw(uint16_t iter, uint16_t n_max);
   rgb565 iter_to_grayscale(uint16_t iter, uint16_t n_max);
   int ilog2(unsigned x);
   rgb565 iter_to_colour(uint16_t iter, uint16_t n_max);
   rgb565 iter_to_colour1(uint16_t iter, uint16_t n_max);
   ```

4. Copy the file `main_flpt.c` to the file `main_fxpt.c`. Modify the required functionality in `main_fxpt.c` to perform the fixed-point Mandelbrot algorithm.
5. Copy the file `compile_flpt.sh` to the file `compile_fxpt.sh`. Modify the file `compile_fxpt.sh` such that it compiles the fixed-point program.
6. Test your new program on the virtual prototype. What can you observe?

Hints:

1. When you multiply two fixed-point numbers, how many bits do you require, and where is the decimal point?
2. There is no detection of overflow and underflow, how to handle these aspects?

2.3 Your own floating point format

In this second exercise you are going to define your own single precision floating point format. You are free in your choice, however, execution time of your Mandelbrot algorithm should be faster as with the build-in library. The steps to follow are equivalent as those for the fixed-point version.

The minimal set of files should contain `fractal_myflpt.h`, `fractal_myflpt.c`, `main_myflpt.c`, and `compile_myflpt.sh`.

2.4 Grading

The grading consists of:

1. Report: 15%
2. Quality of the code: 15%
3. Fixed-point type used: 15%
4. Working fixed-point algorithm on the virtual prototype: 20%
5. Floating-point type used: 15%
6. Working floating-point algorithm on the virtual prototype (faster execution gives higher points): 20%

2.5 Workflow

This practical work is done in 2 practical sessions, and handed in one week later on moodle. Your submission should contain:

- Your report in `PDF` format (max. 10 pages).
- A `.zip` or `.tgz` file with all the source files (and they need to compile).

2.6 Not a bug, or maybe yes?

If you look at the image of the floating-point implementation, you might observe that at least the first line has some artifacts.... Is this a bug, but where, hardware, my program, we will "maybe" find the source later on in this course, when we know more of the system.