PYNQ-Z2 Mathematically Generated Mandelbrot Images

**1. Provide an overview of the purpose, functionality and features of your PYNQ design**

The aim of our PYNQ-Z2 design project, was to implement the generation of mathematical images using an FPGA and appropriate use of hardware and software working together to make the generation interactive and easily useable by interfacing the PYNQ-Z2 board with Jupyter Notebook.

In terms of functionality, this project generates the Mandelbrot set through the implementation of the mathematical equation (equation 1) shown below:

(1)

* = Represents a complex number in the nth iteration
* = Represents the complex number at the (n+1) iteration
* = Is a constant complex number in the complex plane corresponding to the coordinates of the point being tested.

In this implementation, the equation iterates over points within the complex plane to determine if the point belongs in the Mandelbrot set. This process is repeated for a predetermined value of iterations, which can be set by the user within Jupyter Notebook or until a certain condition is met. In our case, the condition which would have to be met is if the magnitude exceeds a value of 2, resulting in the value not being in the Mandelbrot set. Otherwise, the magnitude stays grounded and is colored accordingly.

Furthermore, the Mandelbrot fractal can be generated using different variables which can be performed by the user on Jupyter Notebook. For example, changing the resolution of the generated image, the dpi and the number of iterations, essentially setting the accuracy of the Mandelbrot fractal, as well as the colourmap for the plot.

Determining the magnitude of the complex number proved difficult as the square root operator provided by Xilinx operates on floating point numbers only which are not feasible for hardware implementation as it would severely constrain our resources. Of course, it would be feasible to find the magnitude in the Jupyter notebook however we envisioned performing all heavy computation on hardware hence we decided to implement a magnitude approximation. The technique we researched and implemented is the Newton Method [2].

(2)

* = An estimate of the square root of A.
* = The approximation at the (n +1) iteration
* A = is the sum of the squares of the real and imaginary parts of a complex number.

is modifiable to the user in the final design. However, the closer is to the final approximation of the square root the quicker the algorithm finds the answer. Through testing we found that an ideal value for . This is due to several factors. The first assumption was to set as the reason we are finding magnitude is to judge whether the magnitude exceeds 2 as this would indicate the number has become unstable or simply that the complex number is no longer contained in the Mandelbrot set. This worked well for all numbers in the plane which were unstable however did not work so well for numbers which were stable hence their magnitude would rarely exceed 1. Through testing of extreme complex values (stable and unstable) it was found that using the algorithm found an accurate square root approximation to a degree of accuracy of 2 decimal points after 4 iterations. Therefore 4 iterations were implemented in the hardware design.

**2. Review your system architecture, and explain which components are implemented in software and in hardware, and the rationale for your design choices. Please include a block diagram as part.** All the heavy mathematical calculations in our design have been done on hardware, with any decision making and data processing carried out through Python code in the jupyter notebook. We decided this would be the best way to implement the design because mathematical processes are optimal for hardware implementation using logic cells and look up tables, while implementing something like a repeating loop is not as simple. The hardware implementation can be seen in the images shown in the appendix.

The software is designed around 3 key functions, which are as follows:

* The ‘mandelbrot’ Function
* The ‘generate\_mandelbrot’ Function
* The ‘plot\_mandelbrot’ Function

The main body of the code consists of a few editable variables defining: the resolution of the image, the number of iterations a pixel can pass through before being classed ‘stable’, and the region of the mandelbrot set the user wishes to plot in their image. It was decided to make these values variable because they each have a large impact on the image itself. The variability of the values allows the user to produce different images depending on the values chosen.

The code executes the ‘generate\_mandelbrot’ function first. It splits the desired region of the complex plane into a grid with dimensions defined by the resolution variable. Each point in this grid is then passed into the ‘mandelbrot’ function, where it is passed through the hardware repeatedly. The hardware contains one iteration of the Mandelbrot algorithm and an adjacent magnitude approximation for the result of the iteration which is then fed back to the software. The software then judges whether the number is still contained in the set based on the magnitude approximation and makes the appropriate decision to either map the point as unstable or continue iterating. The algorithm will continue iterating either until the point becomes unstable, or until it reaches the user defined maximum iterations, at which point it will deem the number as stable and part of the set.

After sending every pixel through the generate function, the two-dimensional array of results is plotted using a python library. The plots are coloured according to the value of the iteration in which they became unstable. This process is what creates the final image.

The ‘one iteration at a time’ approach to our design ultimately meant we were restricted to using AXI4-Lite interconnection to pass data back and forth allowing us software decision making. For computations of over 10,000 complex numbers (points on the plot), the algorithm required significant amounts of time to run. To speed up the algorithm as much as possible we utilized Block Ram and DSP slices for arithmetic, and bit shifting operations instead of multipliers or dividers wherever possible. Ultimately, we were forced to implement division blocks for the magnitude approximation which add significant delays and are resource intensive. This is why the magnitude approximation was kept to only 4 iterations as it allowed an accurate approximation but maintained the lowest possible strain on resources and speed.

One issue that arose was the realization that python couldn’t read negative values from the board and couldn’t pass real numbers into the IP. This issue prompted the decision to include another function which converts the values to signed and use reinterpret blocks in the design to get around fractional bits. A rough diagram of our full system can be seen below.

A diagram of a computer

Description automatically generated

Figure Block Diagram of System

**3. Describe how you performed testing to ensure correction operation.**

Throughout our Simulink development we implemented functional testing, this involved passing in various inputs of complex values, through our Mandelbrot algorithm and verifying that the outputs matched the expected results at each stage and at the next stage the same process was implemented to test the magnitude approximation. This process simplified troubleshooting by pinpointing where values were misinterpreted, and incorrect outputs occurred.

Furthermore, during our development of the algorithm we encountered various errors, some of these included unexpected behavior of the ‘Divide’ block, conversion issues when using ‘reinterpret’ and ‘convert’ blocks. The testing here involved implementing various values for the delays, changing simulation time.

In order to test the software, it was developed and run initially in visual studio code, using the mandelbrot formula instead of the hardware. This allowed the code to be developed before the hardware was operational. It also ensured that all other aspects of the code functioned as intended.

While developing more complicated segments of code, both in experimentation, and attempts at different approaches, bugs in the code were far more common. To find where the code was going wrong, we utilised a similar method to the hardware testing. By inserting print statements throughout the code, we were able to track the values of certain variables, discovering the exact locations where errors occurred and also found out which parts of the code weren’t being reached at all.

**4. Explain your approach to project management and group working, including the allocation of tasks, scheduling, and any challenges that arose (and how you handled them). On reflection, is there anything that you would have done differently?**

The project was managed well, and we spent a large amount of time working on the various aspects of the project together in person. We played to each other’s strengths with Finlay spearheading our efforts on the software side of things and Maks and Jakub focusing on analyzing the mathematical algorithms we required and implementing them on hardware using System Generator. When it was time to bring the software and hardware elements together, we tested the system and ran troubleshooting as a group.

When reflecting on the implemented hardware design, in order to achieve greater accuracy, it would be possible to implement more advanced magnitude approximations into the design. However, this might prove difficult as many of them may include more divide blocks, or even trigonometric functions. Furthermore, it could be possible to change the subsystems.

Even though a successful implementation of the AXI Stream itself was generated, including corresponding *.bit* and *.hwh* files, we encountered problems in implementing code to interface with the AXI Stream channels. The time constraints and complexity involved in implementing an AXI Stream which interfaces with the Jupyter Notebook caused us to return to an approach which utilised AXI Lite for communication between the FPGA and the Jupyter Notebook.

While AXI Stream offers many advantages in terms of data transfer speed, the Python implementation required for handling high-resolution images with over 100 iterations would have come with significant challenges. In particular, flagging magnitudes which have exceeded a value of 2, and keeping track of the pixel position which would have to be graphed. This would include the creation of a sorting and indexing algorithm which would allow for the data to be replotted correctly, which introduces complexities which exceeded the project’s timeline constraints.

The group decision made was to prioritise the creation of a successful implementation of a Mandelbrot generation using AXI Lite communication for simpler communication and control however this is significantly slower than what an AXI Stream implementation would be and in turn only allows for lower-resolution, and lower-iteration images to be generated within a reasonable time frame.

**5. Reflect on what you have learned from this project, highlighting aspects that have required independent research and investigation.**

Throughout the project we had to do independent research, from researching the Mandelbrot equation itself and how to generate it, to several implementations of magnitude approximation which were simple and accurate enough for our needs.

Also, during the testing stage, the research became a critical point during the project for diagnosing and identifying errors. As an example, we realized that some of our input and output signal errors were happening because of insufficient simulation time. This allowed us to understand what was going on in the simulation when certain values were misread as fractional bits only due to simulation duration, when expecting a whole number with fractional bits and how to fix it in hardware.

A lot of research was also required throughout the implementation of the software. Such as finding efficient methods of converting data types and structuring binary strings in python. The research carried out on this further progressed the AXI Stream system development, despite the eventual discontinuation of this approach.

One thing that became evident as the project progressed was that the zoom function wouldn’t operate on the notebook the same way it did on Visual Studio Code. It would plot the defined region, as intended, however, the resolution was not maintained. After a lot of research into the reason for this, the only possible conclusion was that the notebook’s cell feature was preventing the code from keeping the ‘region’ variable consistent between the result and plot functions. Unfortunately there wasn’t enough time left to find the solution to this problem, but the zoom function is still present.

This project has allowed us to reinforce our FPGA skills of hardware implementations, creation and generation of IP cores as well as the use of AXI Lite interfaces and generations of AXI Stream. It provided an opportunity to explore the PYNQ-Z2 board, in turn becoming more familiar with this new technology. It has also let us become more efficient at troubleshooting and solving problems which may be faced within future development of FPGA projects.

**References and other open-source projects used during research:**

**[1] -**  [**https://github.com/lejhy/RenderingFractalsOnPYNQ-Z1**](https://github.com/lejhy/RenderingFractalsOnPYNQ-Z1)

**[2] -** [**https://github.com/Christian376/Pynq-math-images**](https://github.com/Christian376/Pynq-math-images)