Imperial College London

Department of Electrical and Electronic Engineering

IMPERIAL

Mathematics Accelerator

Second Year Engineering Design Project ELEC50015

Authors:

Gurjan Singh Samra	02288570
Arjan Hayre	02137475
Amin Mohamed	02216105
Dhyey Trivedi	02210861
Harun Aslam	02032283
Alex Seferidis	02269571
Joshua Gavin	02225408

Summer 2024

Marker: Edward A Stott

GitHub Repo Link

Abstract

For the 2024 second year engineering design project, a solution is presented for an educational tool used to visualise complex fractals, aided by an FPGA with synthesised accelerator hardware. In this project an initial SystemVerilog development stack was used to design a parallelised accelerator. This accelerator included a pipelined engine to perform Mandelbrot calculations and an advanced FIFO system to maximise system performance. This design, after thorough testing using multiple testbenches, was synthesised on a PYNQ-Z1 FPGA - making good use of its resources by instantiating numerous parallel engines for the final design.

Additionally, a website UI combined with a server-side database was developed to enhance the system's user-experience. Users can easily update parameters on the website and then quickly see the calculated visualisation on a screen. The server-side database also offers educational functionality along side the UI, to allow students to save images to share with their peers.

In order to integrate the database functionality on the FPGA, a concurrent python program was developed to interface between the database, the synthesised hardware and the HDMI output (for the visualisation). This linked together all the components to communicate with each other, forming the final system.

Finally, extensive testing was performed on ensure that the system requirements were met. This proved to be the case using quantitative and qualitative results, procured using various methods. This project has taught us technical and professional competencies, improved our project management skills and developed our skill-set as engineers.

Contents

1 Introduction **1**

2 Planning and Design **2**

2.1 Planning . . . 2

2.2 Design . . . 5

2.2.1 Overview . . . 5

2.2.2 System Requirements . . . 6

2.2.3 Initial Design . . . 7

2.2.4 Final Design . . . 9

2.2.5 Mandelbrot Set . . . 9

2.2.6 FMEA . . . 10

3 Hardware **11**

3.1 Accelerator . . . 11

3.1.1 Overview . . . 11

3.1.2 Development Stack . . . 12

3.1.3 IP Design . . . 14

3.1.4 Single Engine . . . 14

3.1.5 Pixel Mapper . . . 15

3.1.6 Distributor . . . 16

3.1.7 Combinator . . . 17

3.1.8 FIFO Buffer . . . 20

3.1.9 Colour Look-Up Table (LUT) . . . 24

3.1.10 Fixed Point Multiplier . . . 25

3.1.11 Generating Frames . . . 26

3.2 Optimisations . . . 29

3.2.1 Pipelining the Engine . . . 29

3.2.2 Parallelisation . . . 32

3.2.3 Fixed Point Representation . . . 36

3.3 PCB . . . 36

3.3.1 Overview . . . 36

3.3.2 Circuit Design and Component Selection . . . 37

4 Software **43**
4.1 Website / Front-end . . . 43
4.1.1 Overview . . . 43
4.1.2 General Construction . . . 43
4.1.3 Foundational Elements . . . 44
4.1.4 Mandelbrot Viewport Functionality . . . 45
4.1.5 Mandelbrot Rendered Construction . . . 46
4.1.6 Database Interactivity . . . 47
4.2 Database / Back-end . . . 48
4.2.1 Overview . . . 48
4.2.2 Database Design . . . 48
4.2.3 Endpoints . . . 49
4.3 Embedded Software . . . 50
4.3.1 FPGA-Side Python Client . . . 50
4.3.2 ESP32 GUI / Simulation . . . 52

5 Evaluation and Testing **55**
5.1 Overview . . . 55
5.2 Hardware . . . 55
5.2.1 Software Implementation . . . 55
5.2.2 Hardware Comparison . . . 58
5.3 Software . . . 60
5.3.1 UI - Subjective Testing . . . 60
5.3.2 Website Alteration from Feedback . . . 63
5.3.3 Connection / Transmission Performance . . . 64

6 Conclusion **66**
6.0.1 Overview . . . 66
6.0.2 Future Improvements . . . 66

Bibliography **67**

A Appendix **I**

Introduction

This report details the group's progress through the second year engineering design project. It will cover the planning, design and development stages of the project, to meet requirements. This project comes at the end of the year, bringing each member's combined knowledge from the course together, to create a mathematical accelerator.

The brief of the project is as follows: "Create an educational tool to visualise a mathematical function. The evaluation of the function must be computationally intensive, which means creating a custom-logic FPGA accelerator. To utilise the FPGA fully, requires implementing parallelisation of the algorithm and choosing the best number of representations. Alongside this, a user interface has to be created which will connect to a server. Through this interface, the user can change parameters of the function to manipulate and understand the visualisation."[1]

To successfully accomplish this, the project can be broken down into the following objectives:

1. Create custom hardware, to implement the chosen function, in SystemVerilog and Verilog.
2. Implement parallelisation of the custom hardware to increase performance.
3. Create a front-end user interface to allow the user to interact with the visualisation.
4. Create a back-end server which takes the user's parameters and sends them to the FPGA.

Planning and Design

2.1 Planning

Considerable and effective planning was necessary for the success of this project. Meetings were held regularly to discuss progress and next steps, these can be seen in Tab. 2.1. An agile approach was used throughout the project, separating each week into a defined 'sprint' with desired goals to reach at the end of each one. Each sprint was designed to produce and improve upon a minimum viable product (MVP).

Date	**Purpose**
21/05	Discussion of assigned project option and strategy moving forward
22/05	Problem decomposition, sub-component allocation and basic system design
24/05	End of sprint 1 review
27/05	Start of sprint 2
28/05	Consultation meeting
31/05	End of sprint 2 review and discussion of interim presentation
03/06	Start of sprint 3
07/06	End of sprint 3 review
10/06	Start of final sprint and discussion of strategy for final demo and report
17/06	Report Deadline day
20/06	Demo day

Table 2.1: Record of group meetings

Upon deciding on an agile development methodology, the workflow was designed as follows: The first sprint was allocated to research, team decomposition and setting-up development environments. This enabled efficient and collaborative design of the system.

In the second sprint, with the knowledge and environments in place, teams started working in parallel to design their respective system components. These would later be physically implemented and integrated in the third sprint. The longer, fourth sprint was designed to account for any

problems that halted previous sprints - allowing time to test and improve before the demo.

A breakdown of each sprint can be seen in Tab. 2.2 and a graphical representation in Fig. 2.1.

Sprint	**Goals**
Sprint 1	Planning / Setting Up: • Decompose problem into sub-problems • Assign teams to sub-problems based on expertise • Research relevant fields to enable design • Set-up simulation environment for hardware modules • Set-up necessary software on team member's devices
Sprint 2	Design: • Design a working single engine • Design a parallelised architecture • Design frontend • Design and set-up back-end server / database architecture • Deliver interim presentation
Sprint 3	Implementation: • Integration of designed hardware modules on FPGA • Integration of front-end and back-end • Develop FPGA Python Script • Optimisation of hardware • Improve UI
Sprint 4	Testing: • Full System Testing (Software and Hardware) • Deliver report • Deliver demo

Table 2.2: Record of group meetings

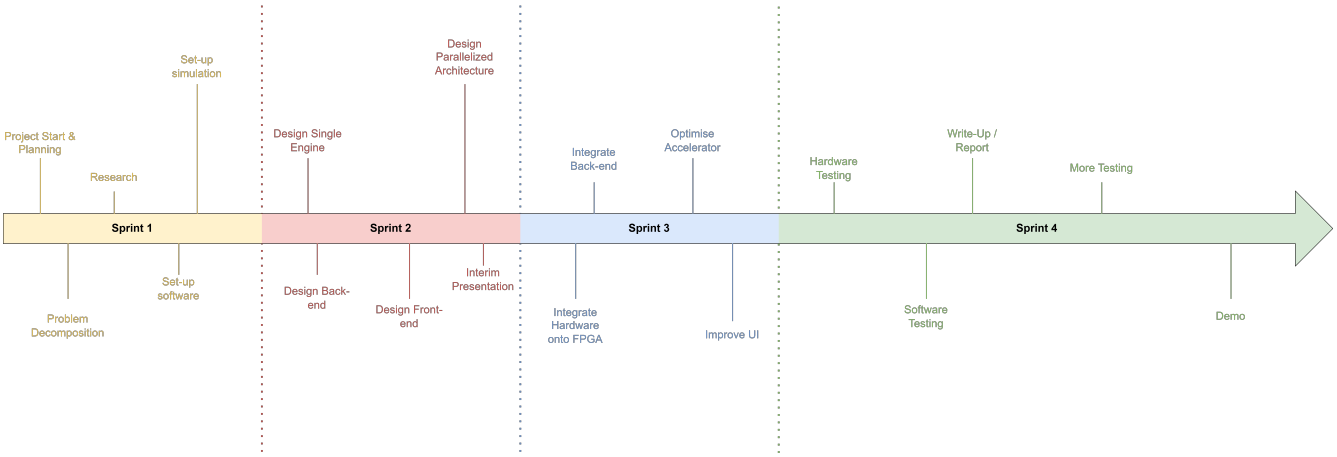


Figure 2.1: Project Sprint Timeline

Allocation of work was based off of a hardware and software team decomposition. Team assignment and individual sub-component designation was decided by experience and expertise. Team member experience was generally informed by their course. All work was organised on a single GitHub repository, allowing easy collaboration between individuals and teams.

Team Member	**Course**	**Team**	**Responsibility**
Gurjan	EIE	Hardware	FIFO Buffer and Combinator
Amin	EEE	Hardware	Distributor and Pipelined Engine
Dhyey	EIE	Hardware	FIFO Buffer and Pipelined Engine
Arjan	EIE	Software	Database / Back-end Development
Josh	EIE	Software	Testbenches and Front-end
Harun	EEE	Software	Front-end Development
Alex	EIE	Software	Integration and Testing

Table 2.3: Responsibility Division per Team Member

A full breakdown of task allocation to group members can be seen in Tab. 2.3. Notably, as the project progressed and tasks were completed, these responsibilities were subject to change and available team members helped finish other components of the system. Task allocation was also informed by task difficulty and importance, for example the parallel architecture requiring two members. The Gantt Chart detailing the planned project timeline can be seen in Fig 2.2.

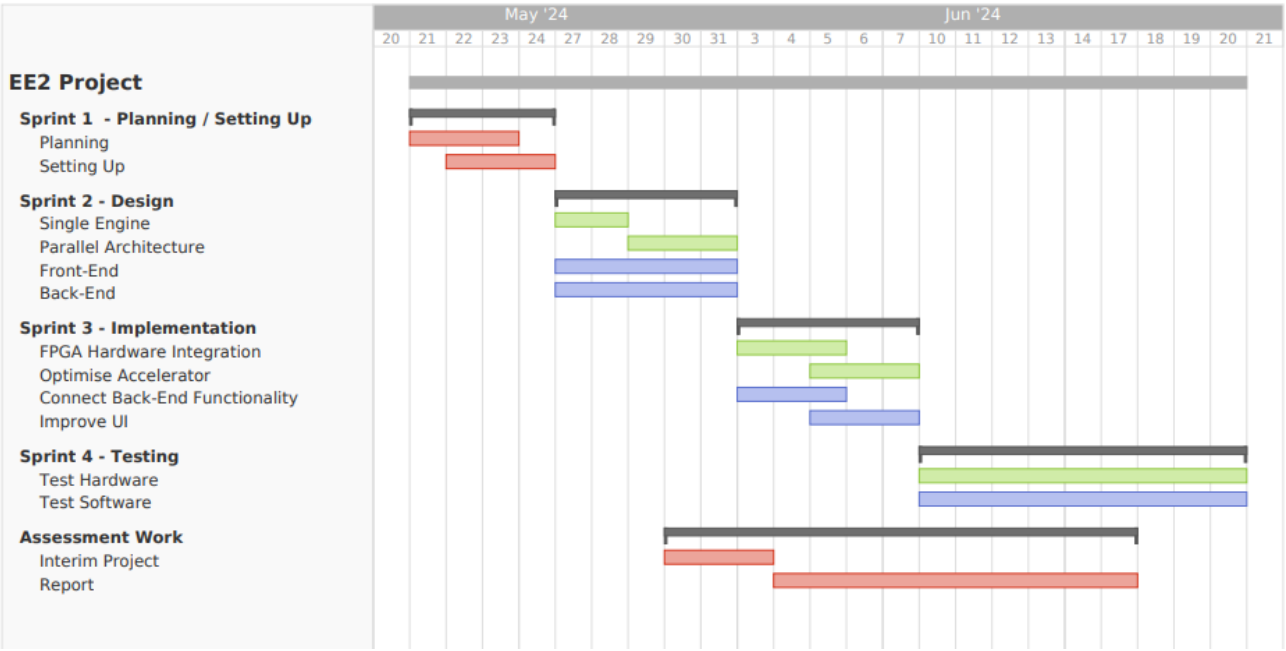


Figure 2.2: Project Gantt Chart [2]

2.2 Design

2.2.1 Overview

The design process was carried out in a formulaic and iterative way. Initially, a strict set of requirements were defined to achieve a desired solution. These requirements informed design and development throughout the project. Additionally, principles such as the aforementioned agile methodology, parallel workflow and frequent meetings / consultations were paramount to satisfying these requirements.

The requirements were concretely defined following several meetings during the first sprint. Discussion was necessary to ensure that the entire team had the same vision for the educational tool. The requirements were set out to guarantee the system was fundamentally accessible, interactive and fun; recognising the system's application and intended user-base.

Once the defined requirements were satisfied, iterative design / development could begin. Therefore, from sprint two onwards, consistent reflection and analysis on whether requirements were on track to be met was necessary.

2.2.2 System Requirements

As mentioned previously, before development could begin requirements were established. These can be seen in Tab. 2.4. These were based off the project brief/guide [1] and team discussion.

Requirement	**Description**	**Testing**
Computationally Intensive Visualisation of a Mathematical Function	The mathematical function used must benefit from a hardware acceleration and optimisation of calculation via the FPGA	Develop a software implementation of the function visualisation. Decision of whether a hardware acceleration is necessary can be made based on observation of processing time
Real-time, Hardware Accelerated Computation	The system should be able to receive parameters, compute a result and realise the output of the computation extremely quickly ($\approx X$ sec)	Evaluate the time taken for the hardware to receive input data and then produce result
	Accelerator hardware design should make use of optimisation techniques to improve performance	Using the same evaluation, determine whether performance improves with the optimisation technique integrated
UI to Enhance the Function of the Visualisation as an Educational Tool	Provide an easy to use UI to allow users to adjust parameters	These can be tested using objective and subjective testing. Objective entails ensuring all functionality works as intended. Subjective testing is performed via user testing and questionnaires
	The UI must be accessible, intuitive and fun to use to bolster the educational experience	
	There should also be server / database functionality to bolster the educational experience	
Reliable / Seamless Transmission and Integration of System Components	Connections between subsystems and devices should be fast and reliable as to not impact user experience	Measure and evaluate the latency of connections between system components. Also observe that data is not altered unexpectedly during the integration and transmission process

Table 2.4: Defined Project Requirements

2.2.3 Initial Design

The initial design to satisfy the requirements defined in Tab. 2.4 was a portable, handheld solution housing the PYNQ FPGA board, a PCB with buttons and two digital five-pin thumb sticks and an ESP-32 micro-controller. These interfaced together as shown in Fig. 2.3.

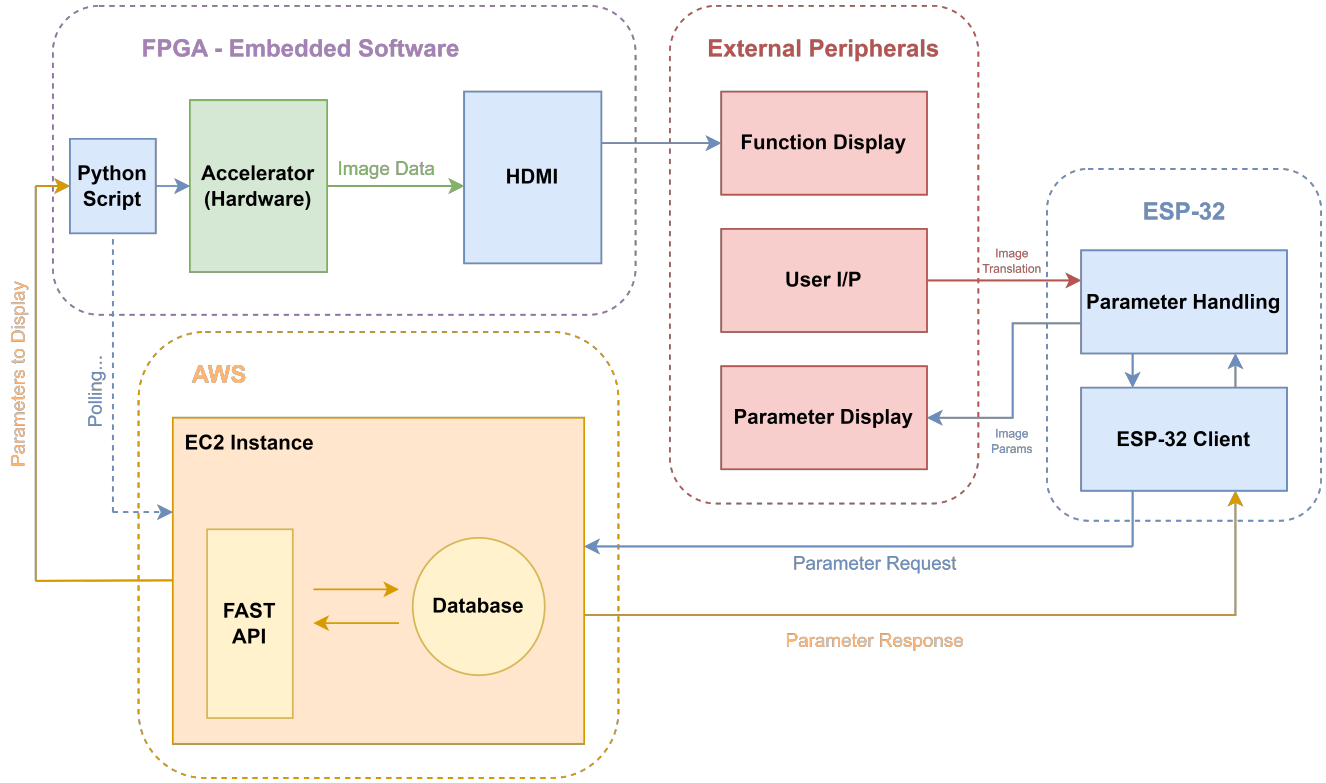


Figure 2.3: Abstracted Diagram of Initial Design

In this design, the user (or student) interacts with the Mandelbrot visualisation using a 'game' controller to change parameters. This input is handled by the ESP32, then sent to the database to be stored as input parameters. These parameters are retrieved FPGA-side, where they are then sent to the accelerator hardware which finally outputs the visualisation via HDMI. Additional database functionality allows for users to save and share parameters which can then be loaded via a menu on the ESP32.

As seen in Fig. 2.4 this design is focused on portability, accessibility and ease-of-use, catering well to its target demographic of students. However constraints in PCB cost as well as long delivery time for components, meant that this design was deemed too risky to pursue.

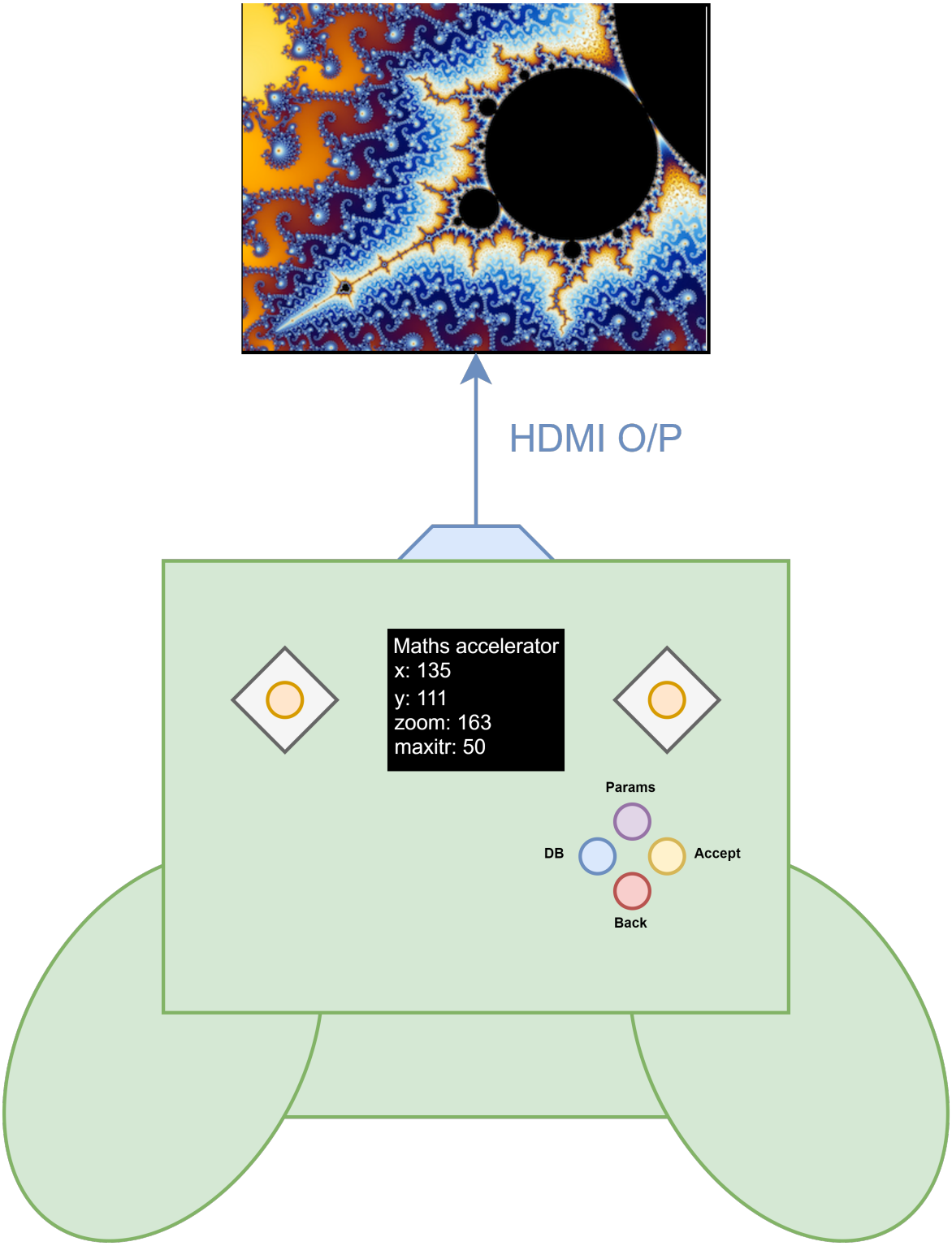


Figure 2.4: Desired Final Implementation of Initial Design

2.2.4 Final Design

The final design, can be seen in Fig. 2.5. Acknowledging the time and cost limitations of the project, the PCB and ESP32 from the initial design were replaced with a web-based implementation for user interaction.

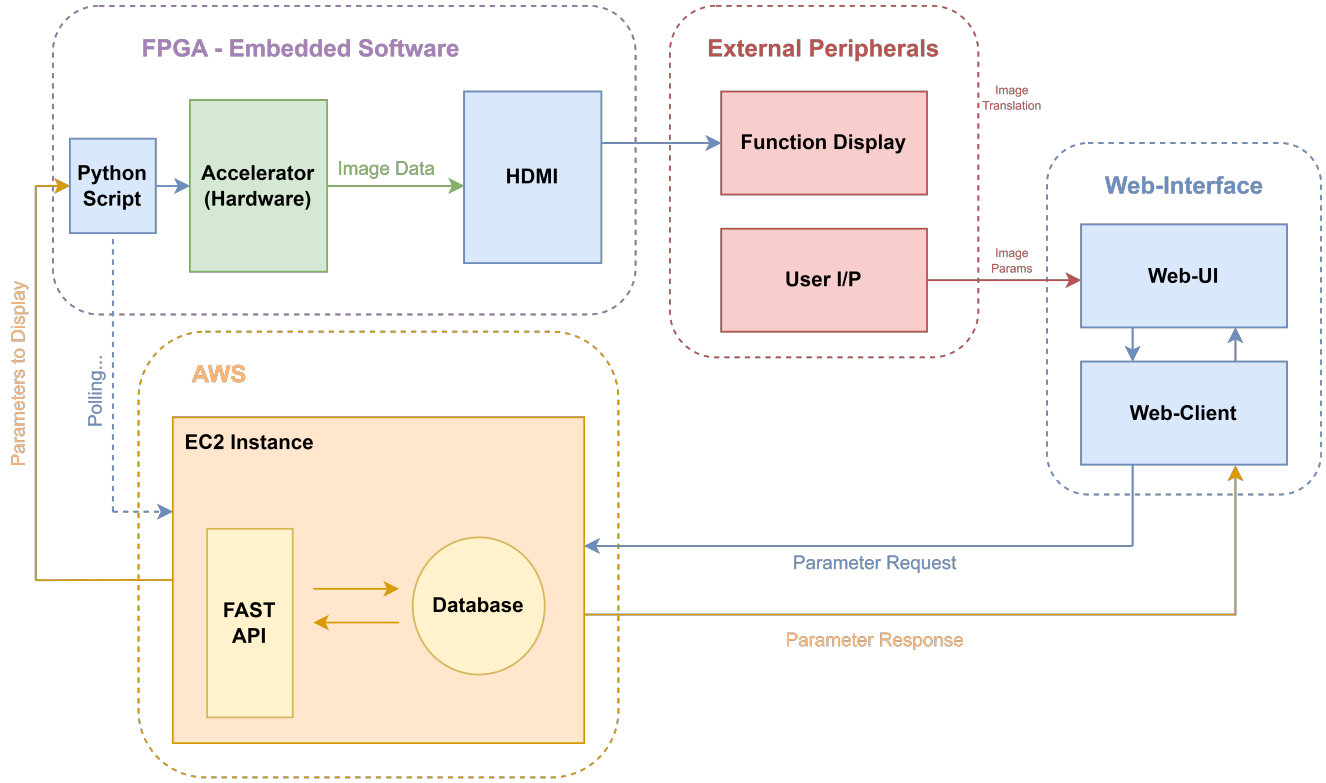


Figure 2.5: Abstracted Diagram of Final Design

This design still provides an intuitive solution for an educational tool, while simultaneously being cost-effective and fast to develop. This implementation also worked in favor with the group's experience and expertise, with multiple members having experience with languages and environments needed for web-development.

Additionally, this solution would allow easier testing and evaluation of the system requirements due to simpler integration between subsystems. This solution also provided greater freedom in satisfying the defined UI requirement, given the available tools for aesthetic manipulation.

2.2.5 Mandelbrot Set

As a group the Mandelbrot set was chosen as it is a compelling visualisation, whilst also being a great educational tool.

The Mandelbrot set is defined by a simple formula: $z_{n+1} = z_n^2 + c$, where z and c are complex numbers. Despite the simplicity of the formula, the set produces a complex and infinitely detailed graph. This is ideal to show the power of the mathematical accelerator, as the accelerator can render high quality images in real time for various zoom levels. The Mandelbrot set is a great example of fractal geometry, which means that as you zoom in, you encounter more intricate structures. This fractal property serves as a perfect demonstration of the mathematical accelerator's ability to compute complex structures at high speeds. Producing the Mandelbrot set involves calculating whether each coordinate in the complex plane is in the set or not. Each coordinate of the Mandelbrot set is independent which is perfect for parallelisation, which is inherently the concept behind the mathematical accelerator. The Mandelbrot set is a well-known example in both mathematics and computer science, since it can be used to model complex theories such as 'chaos theory'. This makes the Mandelbrot set a great educational tool allowing students to bridge the gap between theoretical concepts and practical applications.

2.2.6 FMEA

Included in the appendix (Fig. A.1), a Failure Mode and Effects Analysis was carried out for the development of the video Processing and HDMI (Tx) subsystem as the two were tightly coupled. Throughout the process, problems were systematically identified before they occurred and were dealt with as shown in the appendix.

Hardware

3.1 Accelerator

3.1.1 Overview

The accelerator is the primary component of the system. It is responsible for the computation of the Mandelbrot Set. The accelerator has been designed for parallel processing of the pixel calculations - this is to optimise the system throughput. A detailed diagram of the accelerator design can be seen in Fig. 3.1. This design was implemented on the PYNQ board to maximise computational throughput and take advantage of programmable logic (PL).

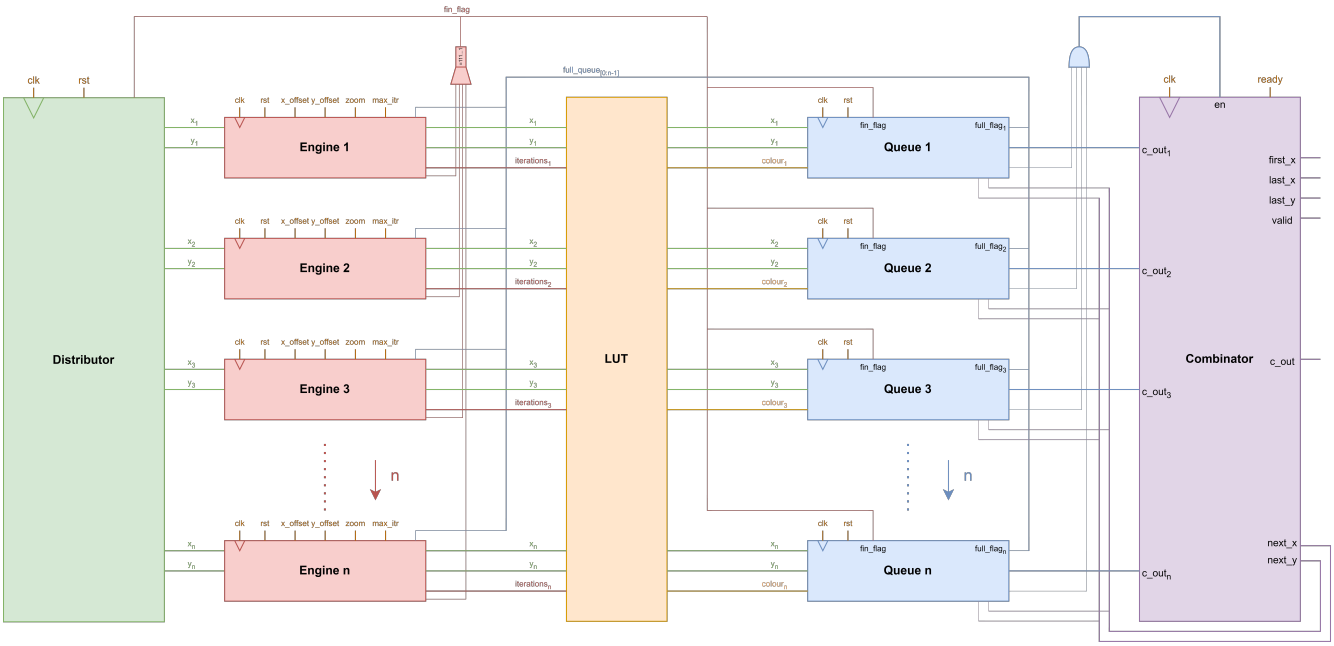


Figure 3.1: Accelerator Hardware Design

In order to improve code readability, the hardware team employed a modular approach. This also minimised code duplication (e.g. the fixed point multiplier module is used thrice in a single iteration).

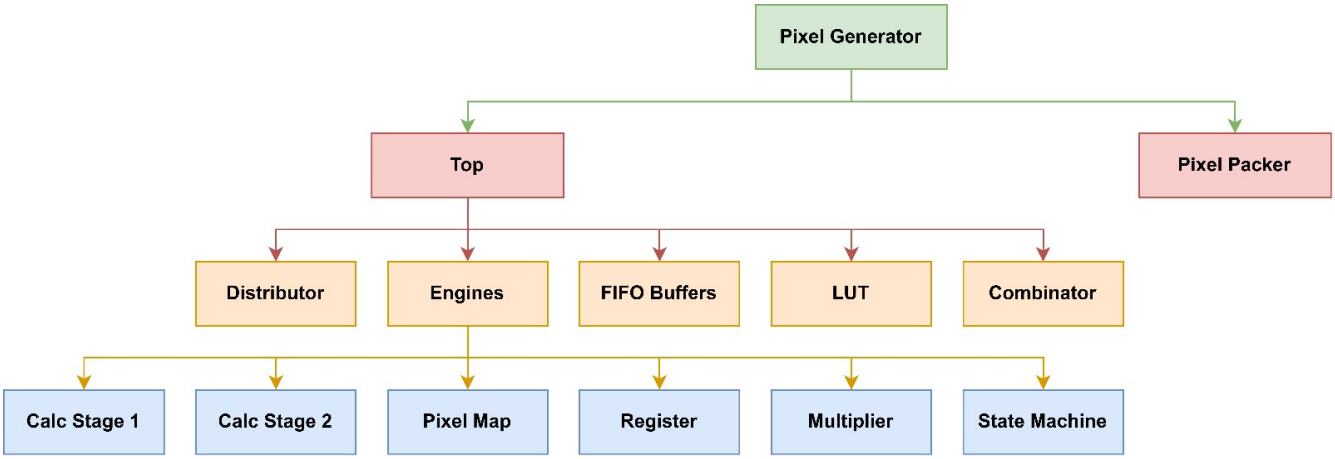


Figure 3.2: Hierarchical Structure of the Hardware Design

3.1.2 Development Stack

To develop the design of the accelerator, Verilator was used to simulate the hardware. This testing method yields results quickly, since generating a bitstream in Vivado can be a lengthy process. In combination with a shell script, Verilator can integrate with our Python image generator to swiftly produce an output image.

In order to debug specific behaviours in the image generation, the VS Code extension WaveTrace was used. This extension provides a cycle-by-cycle signal analysis environment from the *.vcd* file generated by Verilator. Investigation of state changes was then used to identify bugs in the accelerator during development. Fig. 3.3 shows an example of what debugging the accelerator looks like.

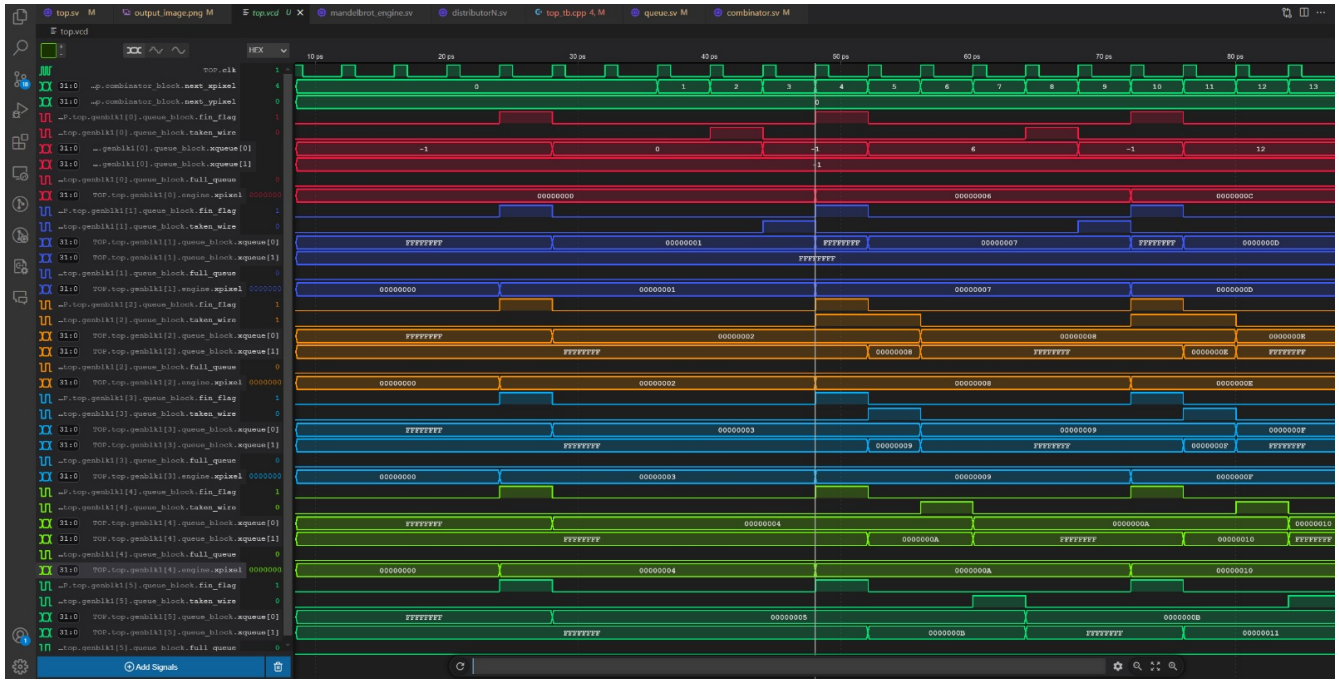


Figure 3.3: WaveTrace Debugging of Parallel Queue Components

The design flow using the above development stack is as follows:

1. Individual .sv files are developed with a top.sv file to link all other hardware.

2. A .cpp testbench file is used to provide appropriate stimuli to the top.sv hardware.

3. Verilator is ran on the testbench and top-level hardware files. This produces a .data file containing the bitmap format HEX colours of the output image. It also produces a .vcd waveform file for later analysis.

4. Using a shell file to launch Verilator and its' dependencies, a Python script is ran sequentially on the .data file to produce a viewable .png file.

5. The .vcd file can be loaded in WaveTrace for analysis and debugging. Bugs can be identified from the output image and located using state analysis of the waveforms.

When developing the individual modules, good practice was employed through a black-box principle. Separate members would work on the respective parent and child modules with knowledge only of the inputs, outputs and functionality of each module. This also includes the `C++ testbench` and `top.sv` files. Using this method ensures modules are developed with their core functionality being assumed, making it easier to identify errors and rectify bugs.

3.1.3 IP Design

A custom IP (Intellectual Property) block was developed in HDL to generate a stream output that is fed into the VDMA. A complete custom overlay was designed with the existing base overlay and custom IP block using Vivado's IP integrator to program the Pynq board. From the utilisation report, the final design, which included 12 engines, used ≈90% of the available slices on the FPGA, and ≈50% of the available DSPs (114 DSPs), indicating high utilisation of the available resources.

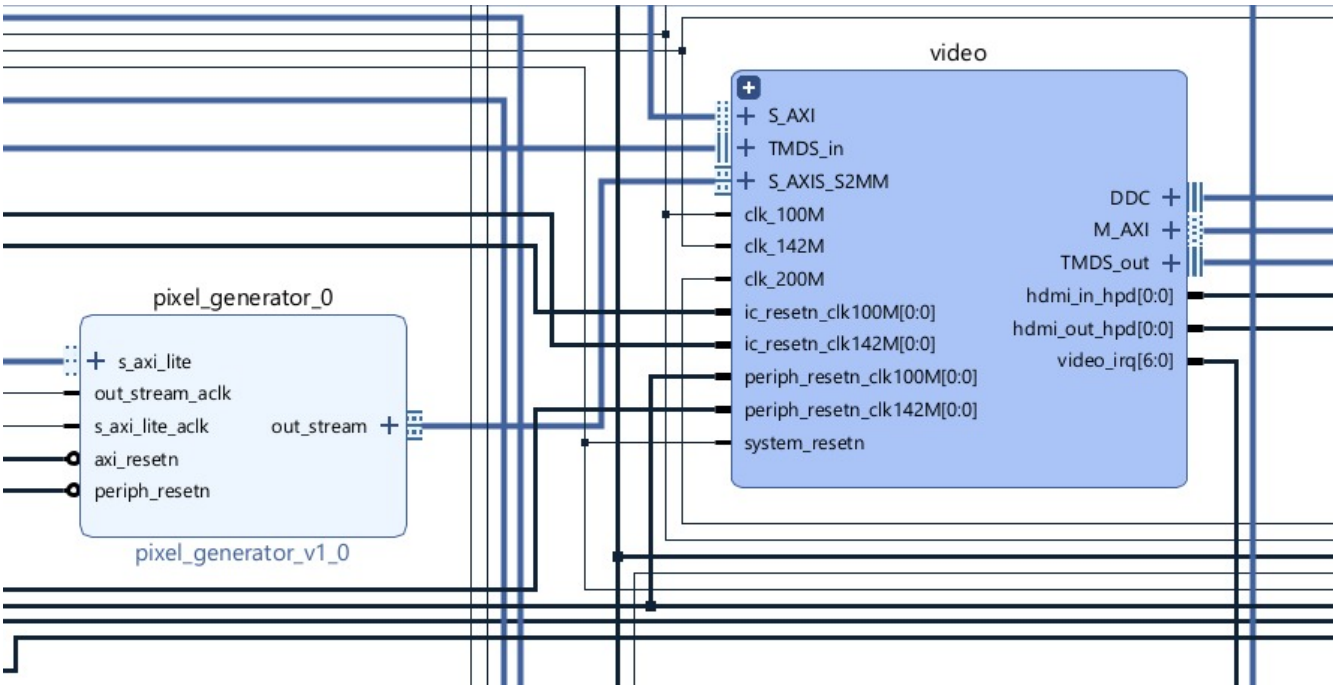


Figure 3.4: Diagram Showing the Custom IP block in the Block Design

3.1.4 Single Engine

The software team's Cython implementation (see Fig. 5.3) of the Mandelbrot set plotting algorithm enabled the hardware team to design a single Mandelbrot engine, which would calculate the number of iterations for each pixel serially. The single Mandelbrot engine would also need to map each pixel coordinate to a complex coordinate, so an algorithm was devised for pixel mapping. The Mandelbrot set is a simple recurrence relation. This recurrence relation is repeatedly calculated for each complex coordinate until the escape condition is met, or the maximum number of iterations is reached.

$$z_{n+1} = z_n^2 + c \tag{3.1}$$

or equivalently:

$$\Re(z_{n+1}) = \Re(z_n)^2 + \Im(z_n)^2 + \Re(c),$$
$$\Im(z_{n+1}) = 2\Re(z_n)\Im(z_n) + \Im(c) \tag{3.2}$$

where c is a constant

The initial design of the single engine involved 32-bit fixed point multiplication (8 bits for the integer part and 24 bits for the fractional part), and the maximum number of iterations was fixed to 50 initially as this design computes each pixel in series. The initial single engine performed all calculations for one iteration in a single cycle. This approach induced a large propagation delay due to the sheer number of calculations required from the engine. Consequently, the engine could not produce the necessary pixel data fast enough to match the VGA clock.

In order to fix this problem, a pipelined approach was used. This allowed the system to run on a higher clock speed and output an image from the FPGA. Further improvements were made, such as parallelisation and buffering.

3.1.5 Pixel Mapper

In order to assign screen pixel values using a quick and robust method, a pixel mapper module was defined. The module takes a screen pixel coordinate alongside zoom and offset variables. From this, it produces a complex coordinate of the corresponding point on the Mandelbrot fractal. This complex coordinate is returned to a single engine to compute alongside the originally inputted screen pixel coordinate. The screen coordinate is passed through, allowing the complex coordinates' iteration count to be attributed to the screen pixel, and hence it's appropriate colour.

```
module pixel_map#(      Cheesypasta1, 3 days ago • 2 engines working but broke
    parameter PIXEL_DATA_WIDTH = 10, //Change as per screen width and height
    parameter ENGINE_DATA_WIDTH = 25,
    parameter ENGINE_FRACT_WIDTH = 20

)(
    input logic clk,
    input logic reset,
    input logic en,
    input logic distributor_ready,
    input logic full_queue,
    input logic [PIXEL_DATA_WIDTH-1:0] pixel_x_in,
    input logic [PIXEL_DATA_WIDTH-1:0] pixel_y_in,
    input logic signed [ENGINE_DATA_WIDTH-1:0] x_offset,
    input logic signed [ENGINE_DATA_WIDTH-1:0] y_offset,
    output logic signed [ENGINE_DATA_WIDTH-1:0] real_x,
    output logic signed [ENGINE_DATA_WIDTH-1:0] imag_y,
    output logic [PIXEL_DATA_WIDTH-1:0] pixel_x_out,
    output logic [PIXEL_DATA_WIDTH-1:0] pixel_y_out
```

Figure 3.5: Overview of Module Inputs and the Parallel Coordinate Outputs

```
multiplier #() M1 (
    .a(scale_factor),
    .b(pixel_x_fixed),
    .rst(1'b0),
    .result(r_gradient)
);


multiplier #() M2 (
    .a(scale_factor),
    .b(pixel_y_fixed),
    .rst(1'b0),
    .result(i_gradient)
);

always_ff @(posedge clk) begin
    if (reset) begin
        real_x <= 0;
        imag_y <= 0;
        pixel_x_out <= 0;
        pixel_y_out <= 0;
    end
    else if ((en)&&(~full_queue)&&(~distributor_ready)) begin
        pixel_x_out <= pixel_x_in;
        pixel_y_out <= pixel_y_in;
        real_x <= r_min + (r_gradient << 6);
        imag_y <= i_min + (i_gradient << 6);
    end
end
```

Figure 3.6: The Functional Code of the Pixel Mapper to Determine Complex Coordinates

3.1.6 Distributor

The distributor is designed to drive each of the engines with the appropriate coordinate. All engines are updated at the same time and they are only updated once all the engines are complete. Therefore all pixels in a group must be processed by the engines before the next are distributed. The pixels are distributed using a raster pattern. This means the pixels are calculated from the top right down to the bottom left. The implementation was designed using the combinational logic seen in Fig. 3.7.

```
always_comb begin
    for(int i = 0; i < NUM_ENGINES; i++) begin
        x[i] = (x0 + i) % SCREEN_WIDTH;
        y[i] = ( (y0 + (x0+i)/SCREEN_WIDTH) % SCREEN_HEIGHT );
    end
end
```

Figure 3.7: Distributor Combinational Logic

The distributor works by storing a reference pixel in a register, this corresponds to the first pixel

from a group of size n (n being the number of engines). The reference pixel is used to determine which coordinates are driven to each engine. This is implemented using the synchronous logic seen in Fig. 3.8. Once the `fin_flag` is asserted (i.e. all the engines are ready to compute the next pixel) the reference pixel is updated and thus, before the next cycle, the engines are driven by the next group of pixels.

```
always_ff @(posedge clk, posedge reset) begin

    if (reset) begin
        x0 <= 0;
        y0 <= 0;
    end
    else if (fin_flag) begin
        x0 <= (x0 + NUM_ENGINES) % SCREEN_WIDTH;
        y0 <= ( (y0 + (x0+NUM_ENGINES)/SCREEN_WIDTH) % SCREEN_HEIGHT ) ;
    end
end
```

Figure 3.8: Distributor Synchronous Logic

The decision to do our calculations in terms of pixels (unsigned integer 10 bits) rather than complex coordinates (signed fixed point 5:20 bits) was quick. This meant that the distributor could leverage integer operations such as integer division and modulo to directly handle the edge case where pixels are distributed over more than one row. Parameterisation of the code using global declarations allowed for greater flexibility in testing.

The distributor was designed to balance the load among the engines. Rather than sending a block of pixels to each engine, the distributor sends a group of pixels that are in close proximity to each other, as their computation times would be similar.

3.1.7 Combinator

The role of the combinator is to output the colour of the pixels in the correct order to the provided pixel packer block. This is achieved by sending the coordinates of the next pixel to the FIFO buffers and forwarding the corresponding colour as an output.

The combinator is controlled by an enable signal and a ready signal. When both of these signals are high, the combinator will output the colour it receives as an input and the next x and y pixel coordinates. The ready signal is an output flag from the pixel generator. When all the buffers' enable flags are high, only then is the enable flag for the combinator high. This ensures that the

combinator always forwards the correct pixel colour every time it is enabled. This was a crucial part of the design as the combinator itself doesn't perform any checking of the pixel coordinates it receives.

The hardware team decided to design the combinator in such a way that it performed minimal computation, in order to minimise the FPGA's resource utilisation. Thus, more engines could be added, increasing speed of computation. Therefore the enable flag was designed to only be asserted when all buffers are full so that the combinator doesn't have to check if a FIFO buffer contains the next coordinate. Although this isn't the best approach, it is simple to implement and has a small trade-off with performance.

The combinator also outputs the last x, last y, valid and first flags required by the pixel packer. The last x flag is asserted when the end of a row of pixels is reached. The last y flag is asserted when the bottom right corner of the screen is reached. The first flag is asserted when the top left corner of the screen is reached (i.e. the first pixel), and the valid flag is high when the combinator outputs a valid pixel colour. The following flags are required by the pixel packer to create a video frame that will be inputted into the AXI VDMA block. Since generating a bitstream in Vivado is a lengthy process, the pixel generator was simulated with a testbench and Icarus Verilog and the combinator asserts these flags correctly.

```
    if((en)&&(ready))begin
        if (xpixel == 0 && ypixel == 0)begin
            first <= 1;
        end
        else begin
            first <= 0;
        end
        if(ypixel >= SCREEN_HEIGHT)begin
            last_y <= 1;
            xpixel <= 0;
            ypixel <= 0;
        end

        else if((xpixel == SCREEN_WIDTH - 1))begin
            last_x <= 1;
            last_y <= 0;
            xpixel <= 0;
            ypixel <= ypixel + 1;
        end
        else begin
            last_x <= 0;
            last_y <= 0;
            xpixel <= xpixel + 1;
        end
        valid <= 1;
        colour_o <= colour_i;
        next_xpixel <= xpixel;
        next_ypixel <= ypixel;
    end
    else begin
        valid <= 0;
    end
end
```

Figure 3.9: Combinator Edge Detection Functionality

3.1.8 FIFO Buffer

The FIFO Buffer was the designed solution to the following problem: The provided pixel packer block, which is used to output pixel colours to the screen, can only take one pixel at a time. This is an issue when there are multiple engines calculating different pixel colour values at the same time. To solve this problem it was necessary to develop a way to store the values and output them one a time in the correct order.

The initial solution suggested was to use a RAM block to hold all the pixel colour values in the correct order. This would be achieved by indexing the RAM using the x and y coordinates of the pixels. A flag would then be asserted when the RAM is full and a counter would output the pixel coordinates in order, one at a time.

Although this solution was simple and easily implementable, it was ineffective for a number of reasons: Firstly the RAM would be too large as it would need to accommodate every pixel on the screen with a 24 bit data width for the RGB values. Another issue would be the writing and reading. These processes take multiple clock cycles, which would slow down the accelerator. Therefore, due to the RAM being inefficient and slow, a better solution was devised.

The improved solution was to use a synchronous First In First Out (FIFO) buffer. The FIFO buffer acts as a queue where the engine outputs the pixel data to the FIFO buffer which stores the data until it is requested by the combinator. The buffer consists of multiple shift registers, a read pointer, a write pointer, a 'full queue' flag, previous pixel checker, an enable flag and a match flag for the combinator (The idea to use two pointers came from [3]).

```
)(
    input logic                     clk,
    input logic                     reset,
    input logic                     fin_flag,
    // received from the engine
    input logic [RBG_SIZE-1:0]      colour_i,
    input logic [DATA_WIDTH-1:0]    xpixel_i,
    input logic [DATA_WIDTH-1:0]    ypixel_i,

    //received from the comibinator to check if front of queue is the next coord
    input logic [DATA_WIDTH-1:0]    xpixel_check,
    input logic [DATA_WIDTH-1:0]    ypixel_check,

    output logic [RBG_SIZE-1:0]     colour_o,
    output logic                    full_queue,
    output logic                    en,
    output logic                    match
);

logic [RBG_SIZE-1:0] colour_queue [QUEUE_SIZE-1:0];
logic [DATA_WIDTH-1:0] xqueue [QUEUE_SIZE-1:0];
logic [DATA_WIDTH-1:0] yqueue [QUEUE_SIZE-1:0];

logic [DATA_WIDTH-1:0]  prev_xpixel;

logic [POINTER_SIZE-1:0] read_pointer;
logic [POINTER_SIZE-1:0] write_pointer;
logic empty_queue;
```

Figure 3.10: FIFO Buffer I/O

The engine sends a finish flag to the buffer when it has finished calculating the colour for the current pixel. The buffer will then check to see if the previous pixel in the buffer isn't the same as the pixel being outputted by the engine. This is crucial since the finish flag from the engine can stay high for multiple clock cycles, as the flag is asserted low only when it receives the next coordinate to calculate. Therefore a check is done to ensure the same pixel isn't added into the buffer multiple times.

Once the buffer has received the pixel data (the pixel coordinates and colour), the write pointer is incremented and enable is asserted. Enable is used to inform the combinator that the buffer has new pixel data which has not yet been read. This can be seen in Fig. 3.11.

```
always_ff@(posedge clk)begin // writes data on the rising edge
    if (reset) begin
        write_pointer <= 0;
    end
    else if (fin_flag && (xpixel_i != prev_xpixel)) begin
        write_pointer <= write_pointer + 1'b1;
        colour_queue[write_pointer] <= colour_i;
        xqueue[write_pointer] <= xpixel_i;
        yqueue[write_pointer] <= ypixel_i;
        prev_xpixel <= xpixel_i;
    end
end
```

Figure 3.11: FIFO Buffer Write Pointer Mechanics

Enable is only asserted when the write pointer and read pointer aren't pointing to the same register. The condition for the buffer to be empty is that both pointers have the same value. This means that the last value that entered the buffer has been read.

The condition for the buffer to be full would also be the same as if it were empty - when both pointers have the same value. These equivalent edge cases present a problem: they cannot be distinguished. To prevent this, the last register is intentionally left empty and never written to. Therefore if the write pointer is pointing to the register just before the read pointer's register, then the buffer is full and the `full_queue` flag is asserted. This can be seen in Fig. 3.12.

```
always_comb begin
    // if the queue is empty
    empty_queue = (read_pointer == write_pointer);
    // if the queue is full
    full_queue = (write_pointer + 1'b1 == read_pointer);

    en = ~empty_queue;
```

Figure 3.12: FIFO Buffer 'Full' and 'Empty' Logic

The buffer size was set to a power of two. This allowed both of the pointers to wrap around to the front of the buffer when they reach the end. Specifically, when the pointer value overflows

(say from 111), it returns to 0. The buffer also has `next_xpixel` and `next_ypixel` as inputs from the combinator. This informs the buffer of the next pixel the combinator needs. If the buffer contains the next pixel data, it will output the colour of that pixel and send a `match` flag back to the combinator. The buffer will then increment the read pointer so that the next pixel data in the buffer will be outputted next. This can be seen in Fig. 3.13.

```
always_ff @(negedge clk)begin // reads data on falling edge
    if (reset) begin
        read_pointer <= 0;
    end
    else if (xpixel_check == xqueue[read_pointer] && ypixel_check == yqueue[read_pointer] && en)begin
        read_pointer <= read_pointer + 1'b1;
        colour_o <= colour_queue[read_pointer];
    end
end
```

Figure 3.13: FIFO Buffer Read Pointer Mechanics

Numerous design decision were made throughout the project to reach the previously described design. Initially the buffer functioned as a queue such that when the front of the queue had been read, the rest of the values in the other registers were shifted up to the front of the buffer. This meant that if the buffer took in and outputted values in the same clock cycle, it would perform read and write operations in the same clock cycle. This was causing problems as the buffer can't perform two dependent operations at the same time, which would cause the buffer to either miss inputs from the engine or to not output to the combinator at the right time. In both cases, the buffer would fail to operate as expected.

The initial approach to fix this problem was to prioritise outputting data to the combinator and create delays for the incoming data from the engine. The delays were created by using wires to store incoming data from the engine before storing that data into the buffer on the following clock cycle. Luckily, the engines never output in consecutive clock cycles so this approach worked. However, storing values on wires uses extra clock cycles and is bad practice. Also using the fact that an engine won't output every clock cycle wasn't a robust solution.

A better solution was designed so that the buffer reads and writes on the rising and falling edge respectively. This saved clock cycles by doing both processes in the same clock cycle. This also means that even if the engine outputs pixels every clock cycle, the buffer is able to handle it.

Although this design works in Verilator, it generated warnings that the counter pointer was being driven twice in one clock cycle. This worked fine for the simulation but was not handled well by Vivado, generating errors. To fix this, the aforementioned, separate read and write pointers were implemented. In place of a single counter pointer, the two pointers turned the basic queue into a buffer (circular queue), where the pointers wrap around once they reach the end. This solution

fixed the problem of the buffer trying to do two operations in the same clock cycle and meant that the design was robust enough to take readings in every clock cycle.

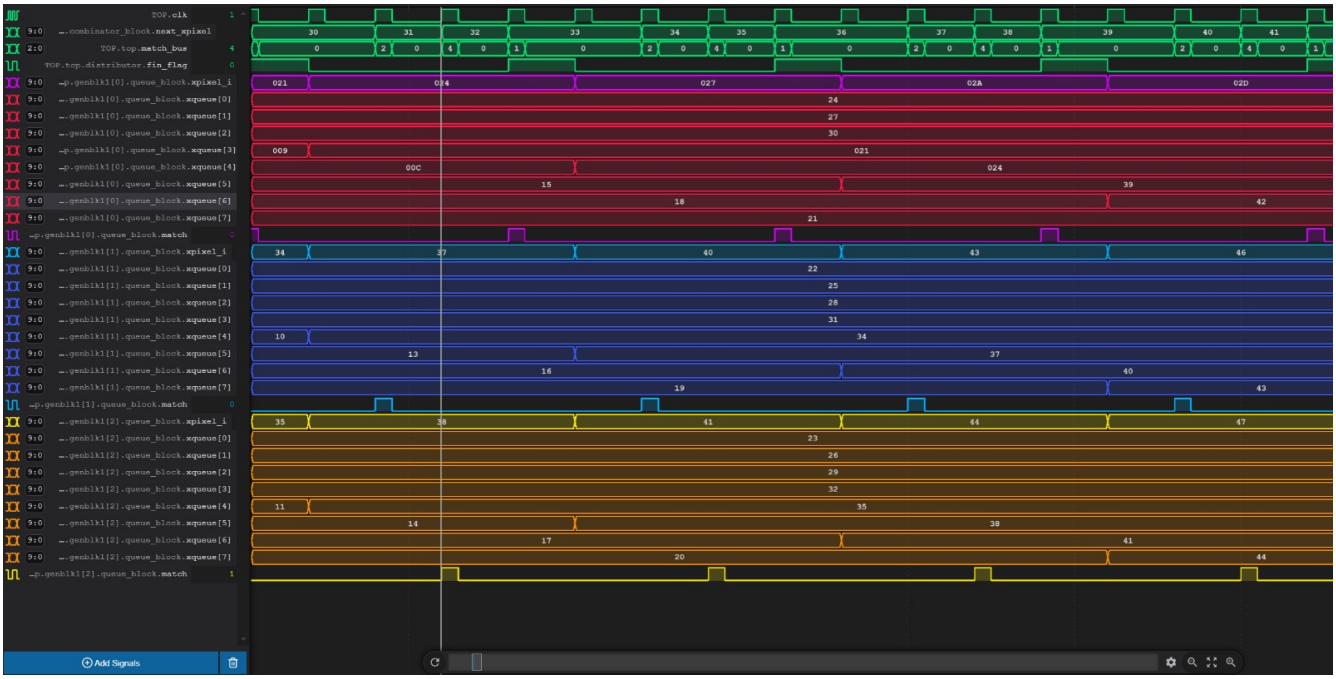


Figure 3.14: A WaveTrace Showing a Fully Functioning FIFO Buffer

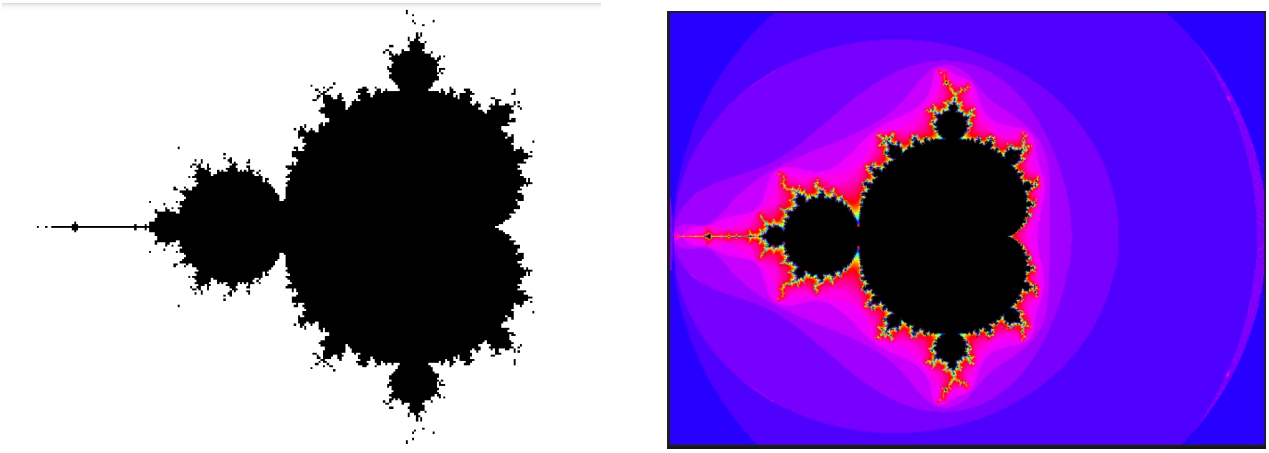
3.1.9 Colour Look-Up Table (LUT)

The LUT is a ROM which stores a full spectrum of RGB values. RGB values are loaded to the ROM on startup. The iterations from an engine are directly proportional to the address of the ROM and hence return an appropriate colour value. The iterations value is scaled by a `step_size` variable determined by the ROM size (256) divided by the maximum number of iterations (assumed less than 256). Hence the ROM address is `iterations * step_size`. This effectively interpolates between the different colours stored in the LUT, always ensuring a continuous colour spectrum. A ternary operator is used to ensure convergent values are displayed in black. The LUT is asynchronous and accepts a bus of iterations to handle multiple engines simultaneously. It then returns the corresponding bus of colours.

```
logic [RBG_SIZE-1:0] lut_array [LUT_SIZE-1:0];

initial begin
    $display("Loading rom.");
    $readmemh("lut.hex", lut_array); // need to create an lut.hex file with rgb values
    $display("rom Loaded");
end
always_comb begin
// always_ff @(posedge clk)begin
    int step_size = (LUT_SIZE / NUM_ENGINES);
    for(int i = 0; i < NUM_ENGINES; i++)begin
        rgb_val[i] = lut_array[(
            iterations[i] == MAX_ITERATION) ?
            (LUT_SIZE - 1) :
            (iterations[i] * step_size)];
    end
end
```

Figure 3.15: Code Snippet Showing Full Functionality of the LUT



(a) Mandelbrot Render without LUT Colouring (b) Mandelbrot Render with LUT Colouring Applied

Figure 3.16: Comparison of Mandelbrot Renders with and without LUT Colouring

3.1.10 Fixed Point Multiplier

A fixed point multiplier was designed since the Mandelbrot engine performs fixed-point multiplications in each iteration. The `product` signal is shifted to the right by the number of fractional bits in our representation, so the correct range of bits is extracted for the final result.

```
module multiplier #(
    parameter ENGINE_DATA_WIDTH = 25,  // Number of integer bits
    parameter ENGINE_FRACT_WIDTH = 20 // Number of fractional bits
) (
    input  logic signed [ENGINE_DATA_WIDTH-1:0] a,  // Fixed-point input a
    input  logic signed [ENGINE_DATA_WIDTH-1:0] b,  // Fixed-point input b
    input logic rst,  // Reset signal
    output logic signed [ENGINE_DATA_WIDTH-1:0] result  // Fixed-point output result
);
    localparam ENGINE_INT_WIDTH = ENGINE_DATA_WIDTH - ENGINE_FRACT_WIDTH;

    // Intermediate signal for the product
    logic signed [2*(ENGINE_DATA_WIDTH)-1:0] product;

    // Perform multiplication
    assign product = a * b;

    // Adjust the product to fit the fixed-point format
    assign result = rst ? 0 : product[ENGINE_DATA_WIDTH + ENGINE_FRACT_WIDTH -1:ENGINE_FRACT_WIDTH];

endmodule
```

Figure 3.17: The Hardware Design for the Fixed Point Multiplier

3.1.11 Generating Frames

The packer module generates frames using a handshaking protocol based on 2 signals - `ready` and `valid`. `ready` is asserted high when the VDMA (Video Direct Memory Access) is ready to receive the next pixel, and `valid` is asserted to high by the combinator when the RGB values inputted into the packer module are correct for the next pixel. The packer module also requires 2 additional control flags which are controlled by the combinator - `first` and `last_x`. `first` is asserted to high when the next pixel marks the start of a new frame, and `last_x` is asserted to high when the next pixel marks the end of a line within the frame.

```
if((en)&&(ready))begin
    if (xpixel == 0 && ypixel == 0)begin
        first <= 1;
    end
    else begin
        first <= 0;
    end
    if(ypixel >= SCREEN_HEIGHT)begin
        last_y <= 1;
        xpixel <= 0;
        ypixel <= 0;
    end

    else if((xpixel == SCREEN_WIDTH - 1))begin
        last_x <= 1;
        last_y <= 0;
        xpixel <= 0;
        ypixel <= ypixel + 1;
    end
    else begin
        last_x <= 0;
        last_y <= 0;
        xpixel <= xpixel + 1;
    end
    valid <= 1;
    colour_o <= colour_i;
    next_xpixel <= xpixel;
    next_ypixel <= ypixel;
end
else begin
    valid <= 0;
end
```

Figure 3.18: The Synchronous Logic Used to Control the Signals Required by the Packer Module

In order to test that the final design follows the handshaking protocol correctly, the frame generator was tested using a Verilog testbench and Icarus Verilog. The flags are asserted correctly, as evidenced in 3.19.

```
for (i=0; i < 3000000; i++){

    for(clk = 0; clk < 2; clk++){
        tfp->dump(2 * i + clk);
        top->clk = !top->clk;
        top->eval();
    }

    top->reset = 0;

    if(top->valid){
        int tmp_index = (3 * index) % arr.size();

        arr[tmp_index] = static_cast<unsigned char>(top->r);
        arr[tmp_index + 1] = static_cast<unsigned char>(top->g);
        arr[tmp_index + 2] = static_cast<unsigned char>(top->b);

        index++;
    }
```

Figure 3.20: Verilator Testbench to Test the Validity of the RGB Values for Each Pixel

```
root@DHYEY:~/EE2-Project-1# vvp stream
Loading rom.
VCD info: dumpfile test.vcd opened for output.
SOF Ok on frame 0
EOL Ok on line 0
EOL Ok on line 1
EOL Ok on line 2
EOL Ok on line 3
EOL Ok on line 4
EOL Ok on line 5
EOL Ok on line 6
EOL Ok on line 7
EOL Ok on line 8
EOL Ok on line 9
EOL Ok on line 10
EOL Ok on line 11
EOL Ok on line 12
EOL Ok on line 13
EOL Ok on line 14
```

Figure 3.19: Results of the Simulation Obtained Using Icarus Verilog

The Verilator testbench ensured that the correct RGB values were inputted into the packer module for each pixel when valid is asserted high. The RGB values for each pixel is written to a 1 dimensional vector in the testbench, which is then used to generate an image using a python script.

3.2 Optimisations

In order to reduce the worst negative slack (WNS), the hardware team optimised the IP design before exporting the design as a bitstream and programming the FPGA.

3.2.1 Pipelining the Engine

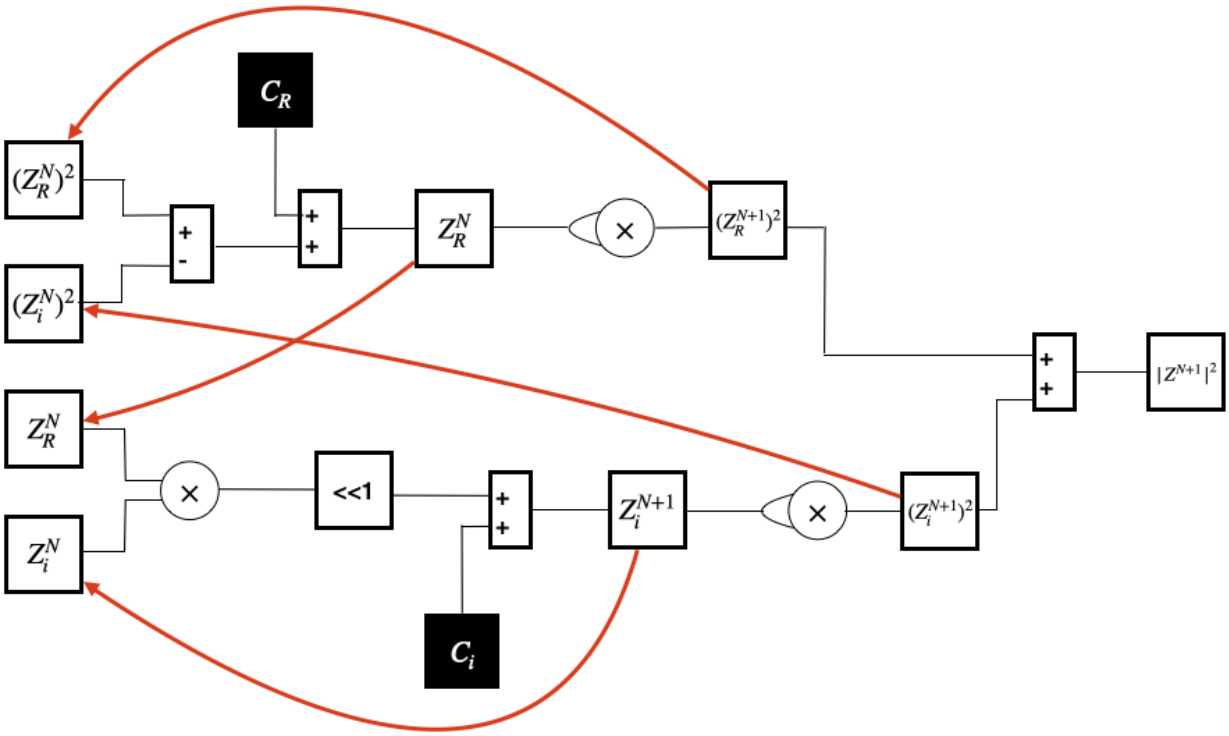


Figure 3.21: Diagram of the Mandelbrot Iteration [4]

From Fig. 3.21, it is evident that each iteration of the plotting algorithm requires 3 multipliers. Therefore if a single iteration of a pixel is computed in a single clock cycle, the critical path of the design would be too large, reducing the maximum clock frequency. This can be resolved by pipelining an iteration, so the mathematical operations can be broken down into multiple pipeline stages. The pipelined engine consists of 4 stages, as illustrated in the state transition diagram in Fig. 3.22.

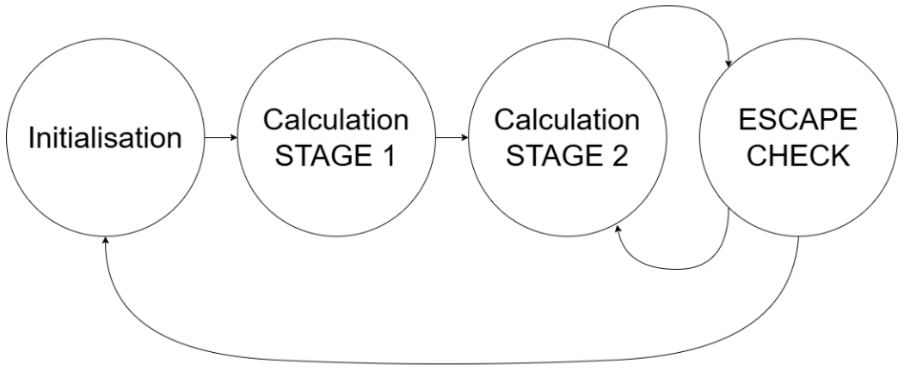


Figure 3.22: State Transition Diagram of the Pipeline Stages

The engine simultaneously computes calculation stage 1 for the next iteration, while checking whether the escape condition has been met for the current iteration in the escape stage. This branch prediction will save 1 clock cycle for each consecutive iteration, reducing the number of clock cycles required to compute pixels within the Mandelbrot set (that do not meet the escape condition). However, if the escape condition is met in the current iteration then this would result in a control hazard, since the pipeline register for calculation stage 1 would hold invalid values for $\Re(z_n)$ and $\Im(z_n)$. This control hazard may be resolved by flushing the pipeline registers before the next complex number (for the next pixel) is computed.

While the number of iterations for the current pixel is being computed in the pipeline, the pixel map stage which maps pixel coordinates to a complex number is stalled until the current pixel has been computed. This means that the Mandelbrot engine can hold the corresponding complex number for the next pixel while the current pixel is being computed in the pipeline.

The control signals required for the pipelined engine can be generated by a mealy machine, described by the truth table below (note that dist_ready is an active low signal). The state transitions are synchronous (implemented in an always_ff block in SystemVerilog), while the output logic will be asynchronous (implemented in an always_comb block in SystemVerilog).

r	**full_queue**	**dist_ready**	**fin**	**state**	**next**	**init**	**en_pix_map**	**en_stg_1**	**en_stg_2**
1	X	X	X	X	INIT	0	0	0	0
0	0	0	X	INIT	CALC_STAGE_1	1	1	0	0
0	1	1	X	INIT	INIT	1	1	0	0
0	0	0	X	CALC_STAGE_1	CALC_STAGE_2	0	0	~full_queue	0
0	1	1	X	CALC_STAGE_1	CALC_STAGE_1	0	0	~full_queue	0
0	0	0	X	CALC_STAGE_2	ESCAPE	0	0	0	~full_queue
0	1	1	X	CALC_STAGE_2	CALC_STAGE_2	0	0	0	~full_queue
0	0	0	1	ESCAPE	INIT	0	0	~full_queue	0
0	0	0	0	ESCAPE	CALC_STAGE_2	0	0	~full_queue	0
0	1	1	X	ESCAPE	ESCAPE	0	0	~full_queue	0

Table 3.1: Truth Table for the Mealy Machine

The enable signals correspond to each pipeline register, and the “initialise” signal is asserted to high when the pipeline registers are flushed. This truth table may be used to design the mealy machine in SystemVerilog.

```
always_comb begin
    case (state)
        INIT: begin
            next = CALC_STAGE_1;
            init = 1;
            en_pixel_map = 1;
            en_stage_1 = 0;
            en_stage_2 = 0;
        end
        CALC_STAGE_1: begin
            next = CALC_STAGE_2;
            init = 0;
            en_pixel_map = 0;
            en_stage_1 = ~full_queue;
            en_stage_2 = 0;
        end
        CALC_STAGE_2: begin
            next = ESCAPE;
            init = 0;
            en_pixel_map = 0;
            en_stage_1 = 0;
            en_stage_2 = ~full_queue;
        end
        ESCAPE : begin
            next = finished ? INIT : CALC_STAGE_2;
            init = 0;
            en_pixel_map = 0;
            en_stage_1 = ~full_queue;
            en_stage_2 = 0;
        end
    endcase
end
```

Figure 3.23: Asynchronous Output Logic for the Mealy Machine

3 pipeline registers are required for the Mandelbrot engine to store data that may be required for each stage, namely the values of z and z^2. A separate module was designed for each pipeline register in order to leverage hierarchical design.

```
calc_stage_1 calc_stage_1(
    .clk(clk),
    .reset(init),
    .en(en_stage_1),
    .zr_next(zr_next),
    .zi_next(zi_next),
    .zr(zr),
    .zi(zi)
);
calc_stage_2 calc_stage_2(
    .clk(clk),
    .reset(init),
    .en(en_stage_2),
    .zr2_next(zr2_next),
    .zi2_next(zi2_next),
    .zr2(zr2),
    .zi2(zi2)
);
```

Figure 3.24: Pipeline Registers for the Calculation Stages

3.2.2 Parallelisation

The initial design of the mathematical accelerator calculates the number of iterations for a single pixel serially. Calculating the iteration value for each pixel is independent, hence utilising multiple engines to compute multiple pixels simultaneously can drastically improve the speed at which the IP can render the fractal. The number of engines that will be used depends on the number of available

DSPs in the programmable logic (220 available DSPs [5]). However, the speed of computation will not exhibit a entirely linear relationship with the number of engines, since the combinator forwards at most 1 pixel in each clock cycle to the packer, so the speed of computation will eventually be limited by the packer module. While this could be avoided by redesigning the packer module to process a row of pixels simultaneously, the hardware team concluded that the number of engines was limited by the number of available slice LUTs, rather than the packer module, which seems to only limit performance when the number of engines exceeds the maximum number of engines that can be programmed onto the FPGA. Fig. 3.25 was obtained by simulating the parallel architecture with a Verilator testbench for a given number of engines. Fig. 3.26 proves that the number of engines is inversely proportional to the number of simulated clock cycles required to produce the Mandelbrot set.

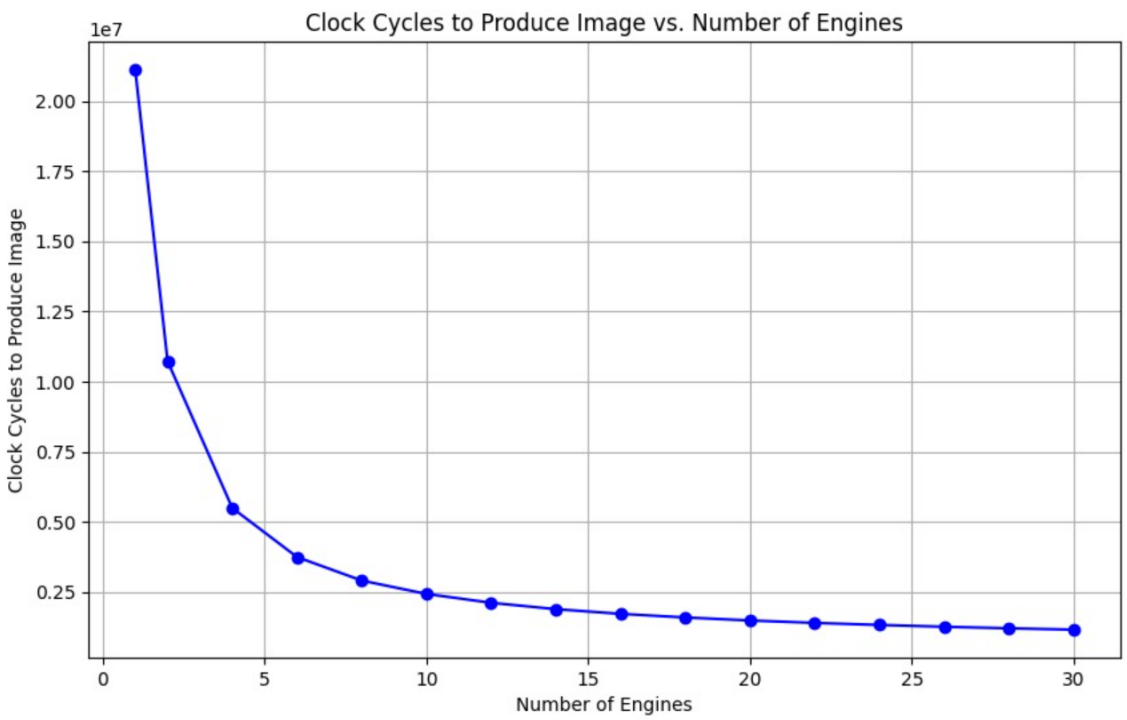


Figure 3.25: A Graph Illustrating the Relationship Between Number of Engines and Number of Simulated Clock Cycles Required to Produce the Mandelbrot Set

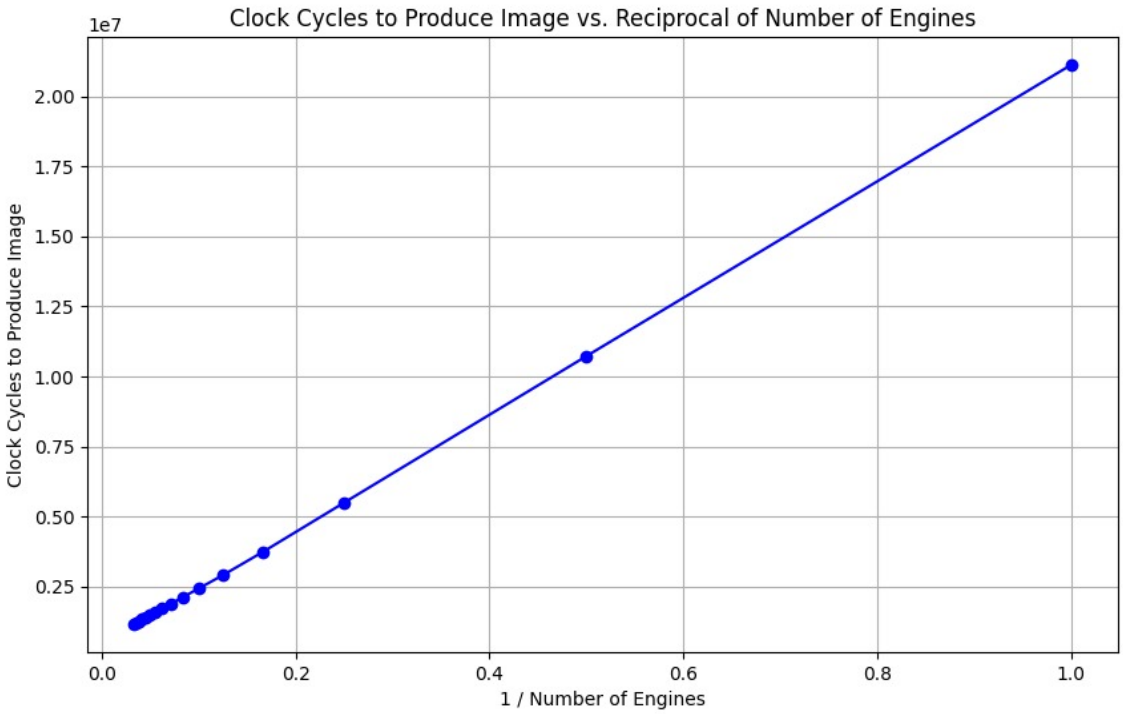


Figure 3.26: A Graph Illustrating the Inverse Relationship Between the Number of Engines and Number of Simulated Clock Cycles Required to Produce the Mandelbrot Set

The `generate` block in SystemVerilog can be used to create multiple copies of the engine, improve the readability of the top level module.

```
generate
    for(i = 0; i < NUM_ENGINES; i++)begin
        mandelbrot_engine engine(
            .clk(clk),
            .reset(reset),
            .iterations_max(iterations_max),
            .x0_(x[i]),
            .y0_(y[i]),
            .zoom(zoom),
            .x_offset(x_offset),
            .y_offset(y_offset),
            .full_queue(full_queue_bus[i]),
            .en_pixel_map(fin_bus[i]),
            .engine_ready(engines_ready[i]),
            .iterations(iterations_bus[i]),
            .xpixel(xpixel_bus[i]),
            .ypixel(ypixel_bus[i])
        );

        queue_new queue_block(
            .clk(clk),
            .reset(reset),
            .fin_flag(fin_bus[i]),
            .colour_i(colour_bus_i[i]),
            .xpixel_i(xpixel_bus[i]),
            .ypixel_i(ypixel_bus[i]),
            .xpixel_check(xpixel_wire),
            .ypixel_check(ypixel_wire),
            .colour_o(colour_bus_o[i]),
            .full_queue(full_queue_bus[i]),
            .en(en_bus[i]),
            .match(match_bus[i])
            );
    end
endgenerate
```

Figure 3.27: The `generate` Block Used to Create Multiple Copies of the Engine

The parallel architecture was tested with a Verilator testbench before generating a bitstream in

Vivado (and determining the optimal number of engines based on the tradeoff between speed of computation and number of sliced LUTs used).

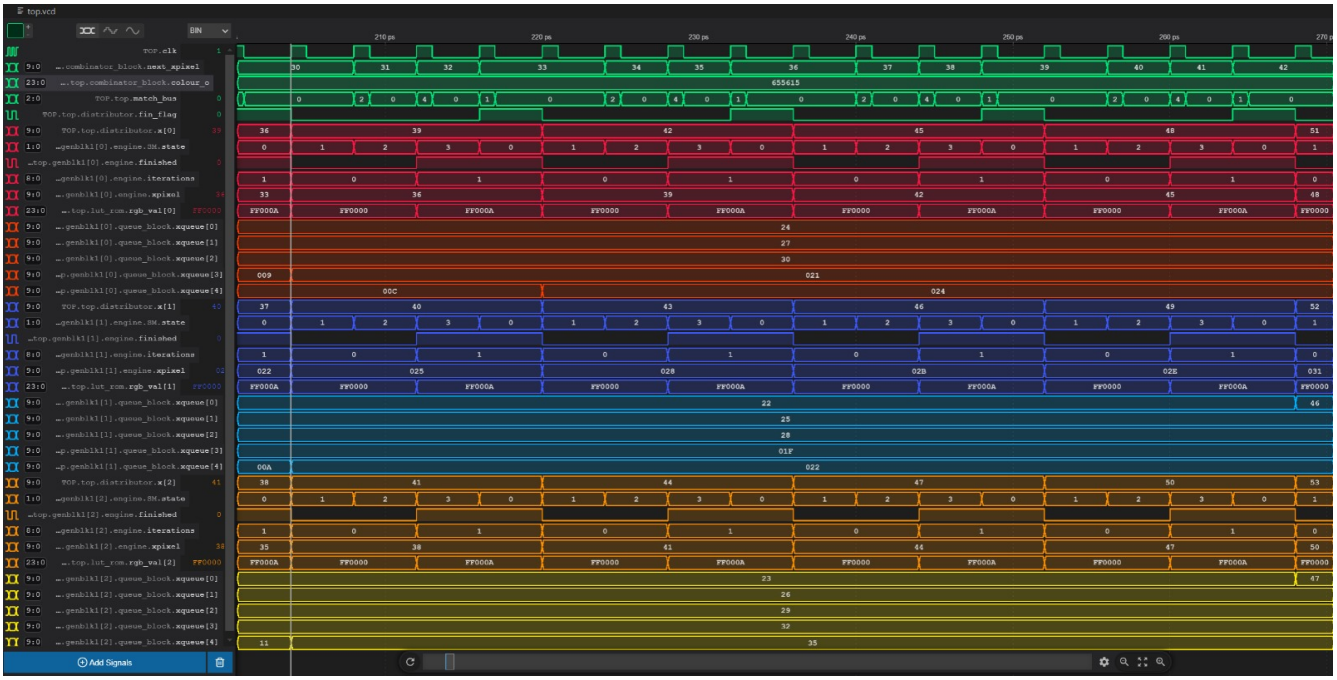


Figure 3.28: A Wavetrace Showing a Functioning Parallel Architecture with 3 Engines

3.2.3 Fixed Point Representation

For our number representation, signed fixed-point arithmetic was chosen over floating-point arithmetic to enhance speed at the cost of accuracy. Given that the Mandelbrot set exists approximately within a circle of radius 4 in the complex plane, an integer bit width of 5 was selected to properly handle the escape condition, and a fractional bit width of 20 for precision.

Additionally, multiplier blocks were chosen instead of division blocks. This choice was driven by the significantly lower propagation delay of the fixed-point multiplier block compared to the division block. Consequently, we were able to shorten the critical path, permitting a higher clock domain frequency for our calculations.

3.3 PCB

3.3.1 Overview

The PCB was responsible for housing the components which handle user interaction in the initial design. The design process was centralised using the EasyEDA platform [6], a PCB design software

used to generate schematics. EasyEDA allowed for an optimised workflow due to its ability to support parallel development. Team members were able to design schematics separately to connect later, making version control management easy. Component selection and budget management was handled automatically by making use of JLCPCB's automated assembly service. This maintained a bill of materials (BOM) which was used in conjunction with an Excel spreadsheet to monitor component costs and links to suppliers. Due to this extensive and efficient budget management, it was quickly discovered that the PCB implementation could not be realised due to cost constraints.

3.3.2 Circuit Design and Component Selection

The components integrated in the PCB design were two, five-way tactile switches, four buttons, an OLED screen, LED indicators and a microcontroller. The microcontroller chosen was the ESP32, primarily dues to its Wi-Fi capabilities. Additional benefits included a built-in screen, buttons and generous flash memory for its cost. Additionally a power switch, voltage regulators, micro-SD card slot, USB-C interface and BOOST/RST buttons were all added for ease-of-development and robustness of the final design.

The schematics for the user controls can be seen in Fig. 3.29, with the five-way tactile switches on the left and the buttons on the right. The five-way switches provide a joystick-like interface allowing the user to select 'up', 'down', 'left', 'right' and 'centre'. Each direction completes a pull-down resistor circuit, providing voltage with current restriction to the allocated GPIO. The buttons also use the same pull-down configuration.

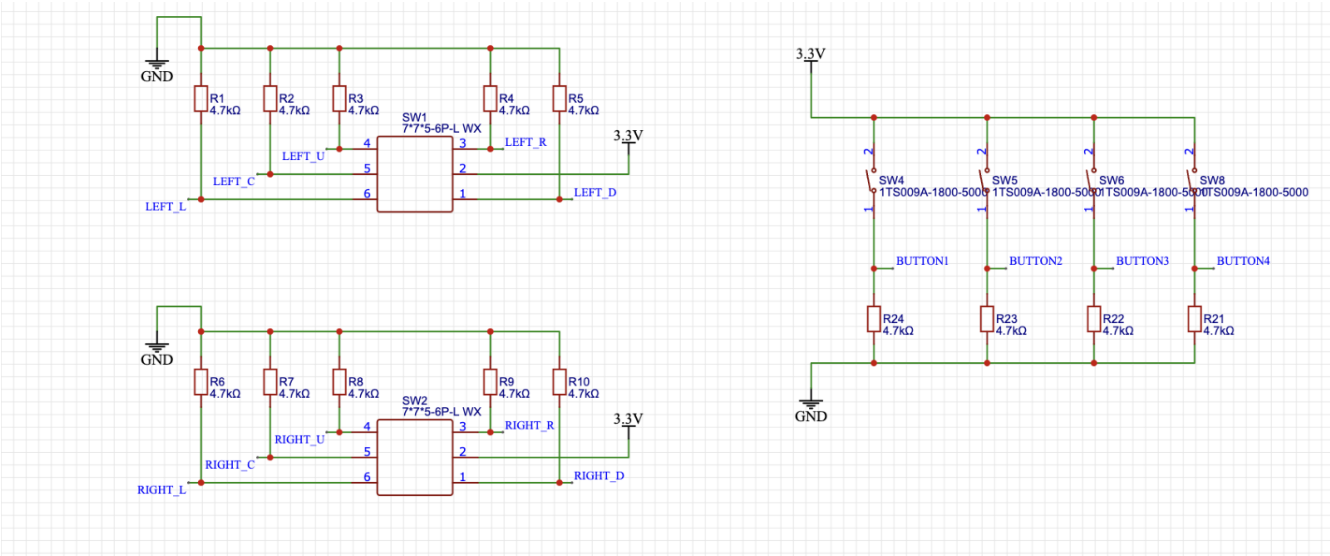


Figure 3.29: Button and Tactile Switch Schematic

The SD card reader schematic can be seen in Fig. 3.30. This included an SD card port (right)

and a logic level shifter (left). The SD card communicates via Serial Peripheral Interface (SPI), which is a reliable protocol for peripheral communication. The logic level shifter provides reliable binary transmission on the SPI channels, ensuring data integrity. The ESP32 interfaces directly with the SPI channels.

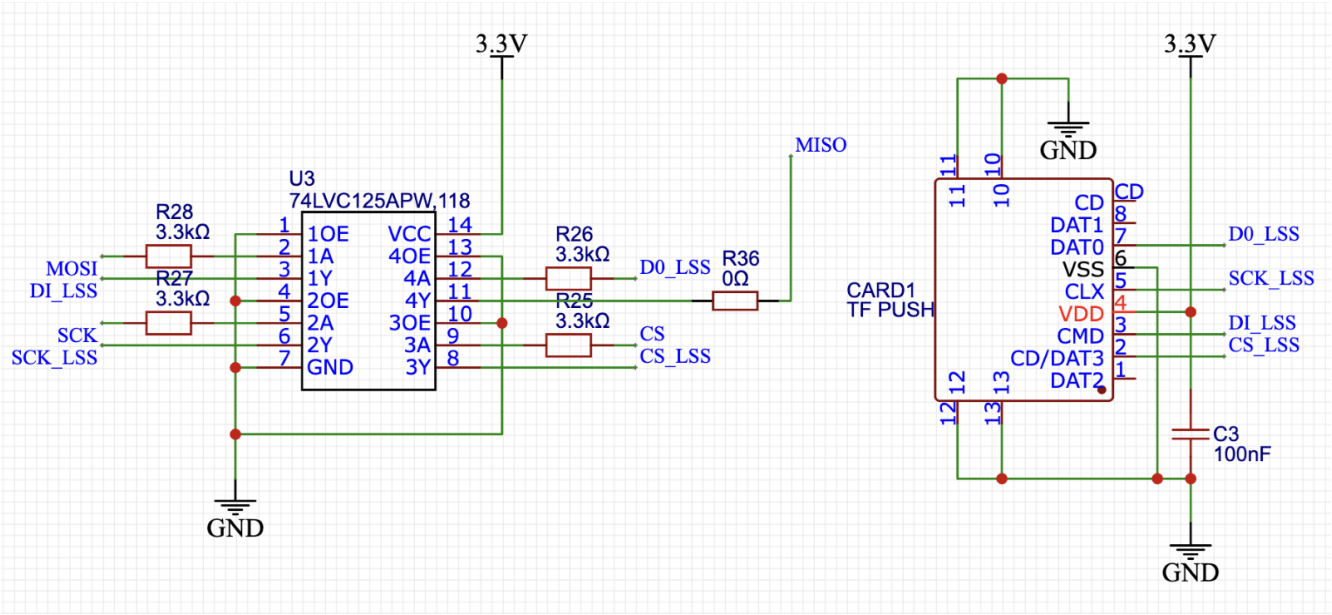


Figure 3.30: Micro-SD Card Reader Schematic

The OLED screen (left) and LED (right) schematics can be seen in Fig. 3.31. The OLED screen connected to the PCB (via a pin header), provides a GUI for adjusting parameters, browsing the database and editing it. The status LEDs are current-limited and connected to the ESP32's GPIO pins, to be assigned roles as needed. These were to be used for debugging and marking of system states.

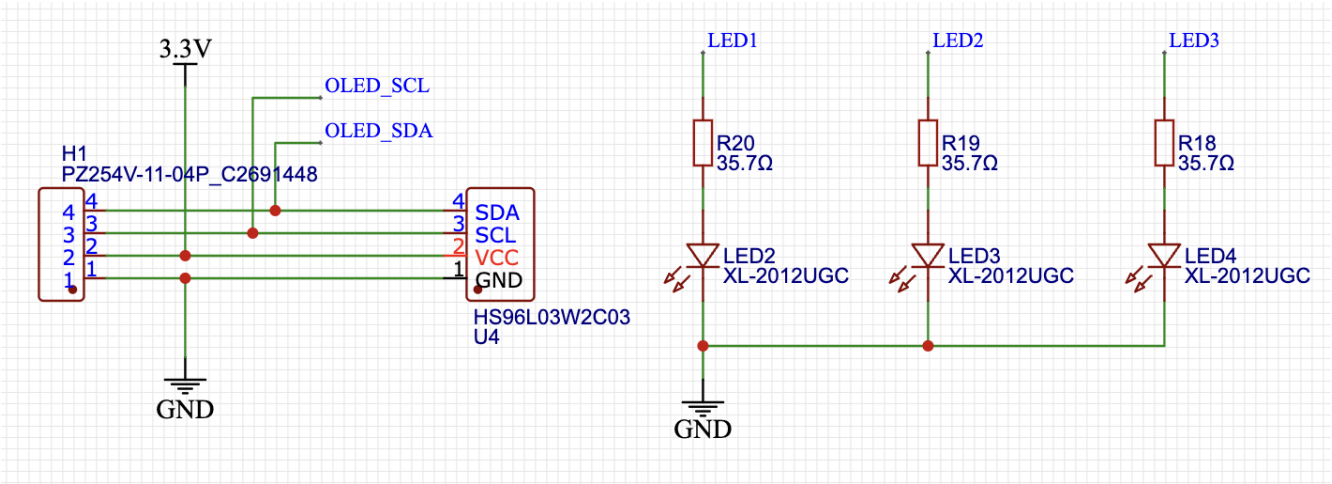


Figure 3.31: OLED Screen and LED Schematic

The power supply schematic can be seen in Fig. 3.32. Pin headers for the PYNQ were a principal feature to the PCB, as it would allow for seamless connection, forming a single handheld device. A was added battery to the VIN pins for the PYNQ to allow for fully wireless capability.

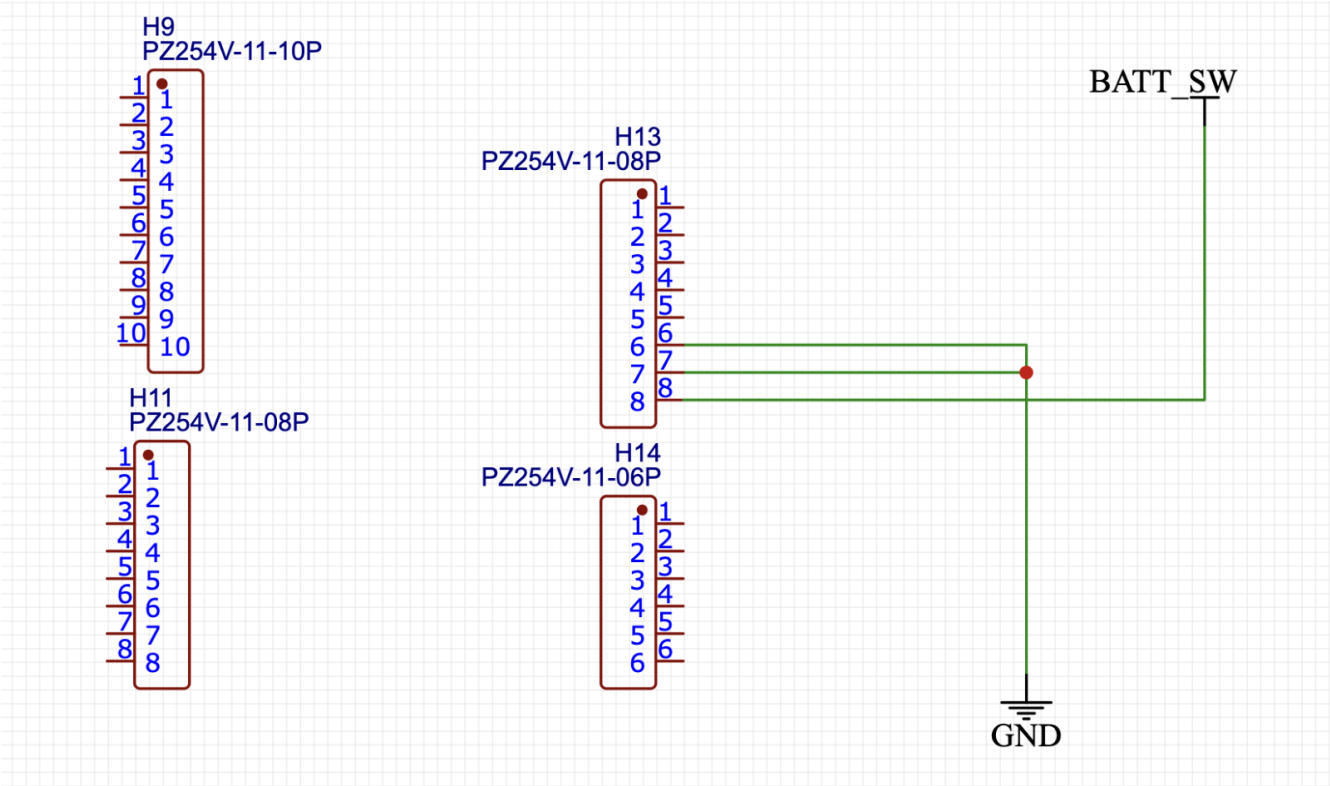


Figure 3.32: Pin Header and Power Supply Schematic

The USB programmer schematic can be seen in Fig. 3.33. It includes a USB-C input (bottom right), USB to UART programmer (top) and USB protection diodes (bottom). The USB-C port was chosen for its commonality and reliability, allowing for easy programming and two-way serial communication. Schottky diodes protected the data channels from voltage spikes. A latching circuit was added for the reset and boot buttons (bottom left) to easily set the programmer chip to the 'program' mode.

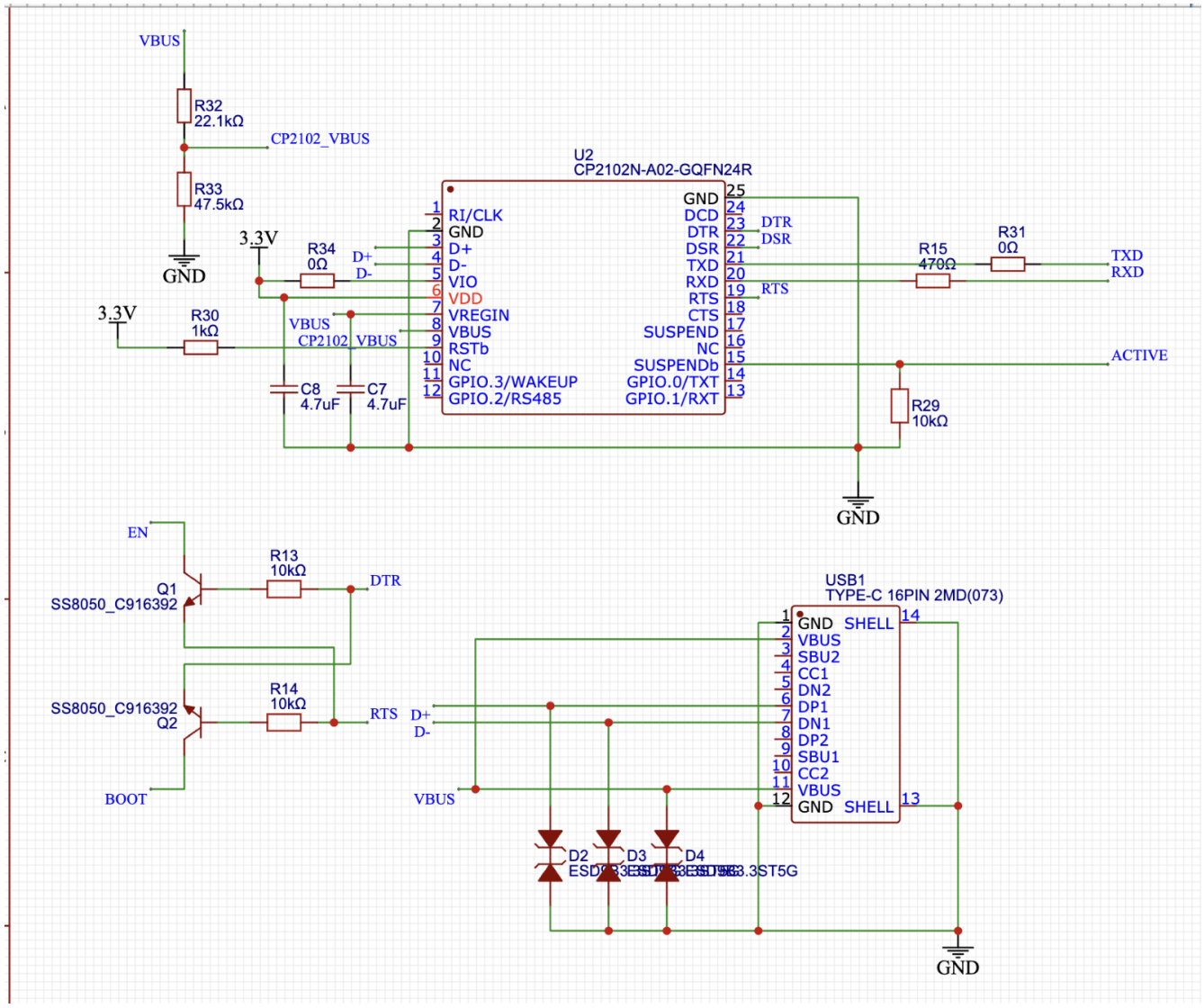


Figure 3.33: USB Programmer Schematic

The power regulator schematic can be seen in Fig. 3.34. This includes a power regulator for both 3.3V and 5V. The PYNQ provides a 3.3V regulator, however the decision was made to use an external regulator. This provided a cheaper solution, which would be less costly to replace if damaged. Zener diodes were used to provide protection from battery connection in reverse-polarity. A switch was added for easy selection of power. Although the 5V regulator was not used in the design, it is cheap and provides the option to add modules in the future.

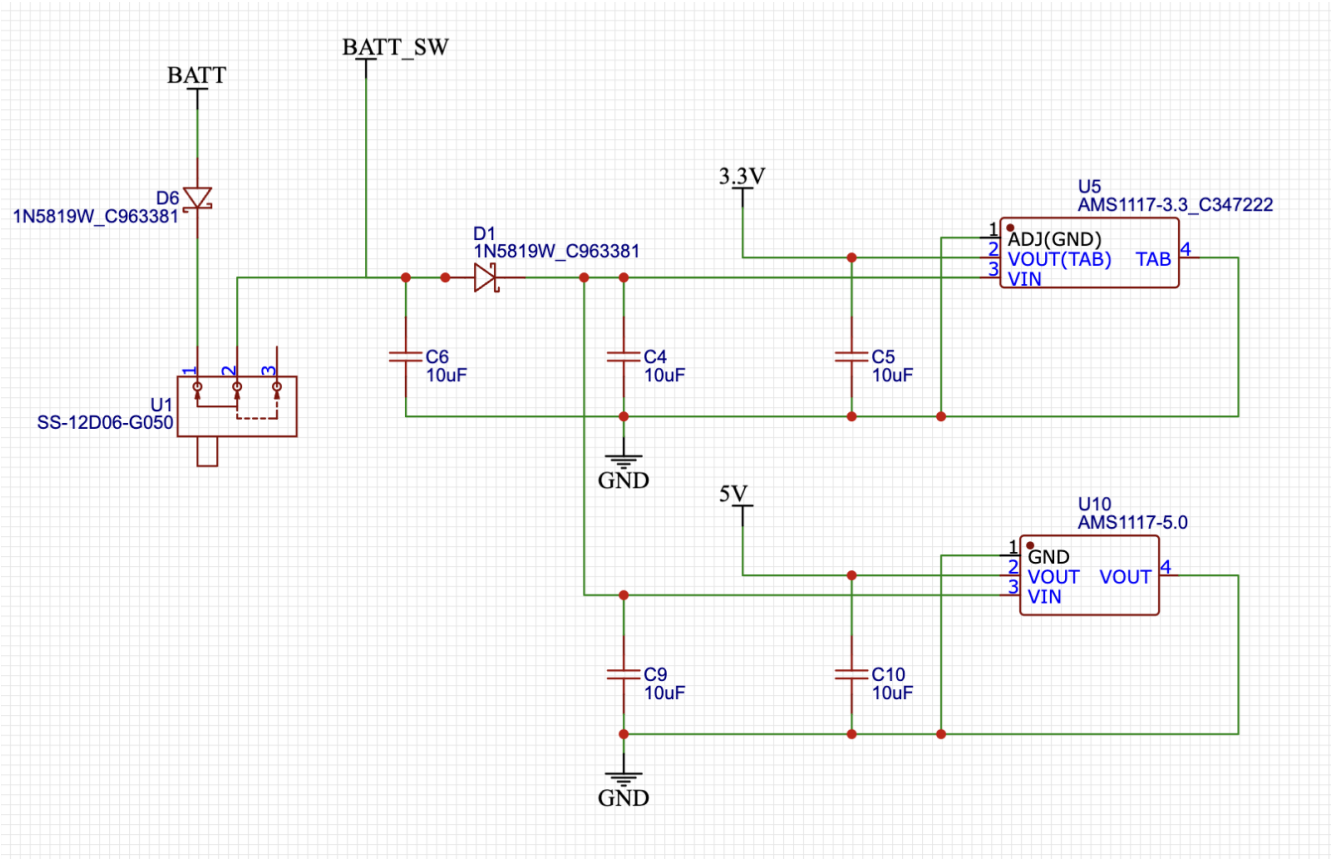


Figure 3.34: Power Regulator Schematic

The top level design connected the nets of all components to the required pins on the ESP32. Fig. 3.35 shows the ESP32 connected to all the defined nets and both the reset and boot buttons (right). The conversion of schematics can be seen in Fig. 3.36. Components are placed on the PCB and ratlines represent the connected nets of the components (see Fig. 3.36a). Traces are drawn manually to connect these ratlines. The layer of each trace is selected from the four available, giving careful consideration to how traces may conflict and interfere on each layer. Good practise is employed to prevent acute angles, which can form pools of etching solution. Clearances are also monitored to prevent obstruction of the Pick-and-place machine during manufacturing. The shape of the PCB was determined by the footprint of the PYNQ, with cutouts for the height of the Ethernet port and jumpers to pass through. The final PCB model following this process can be seen in Fig. 3.36b.

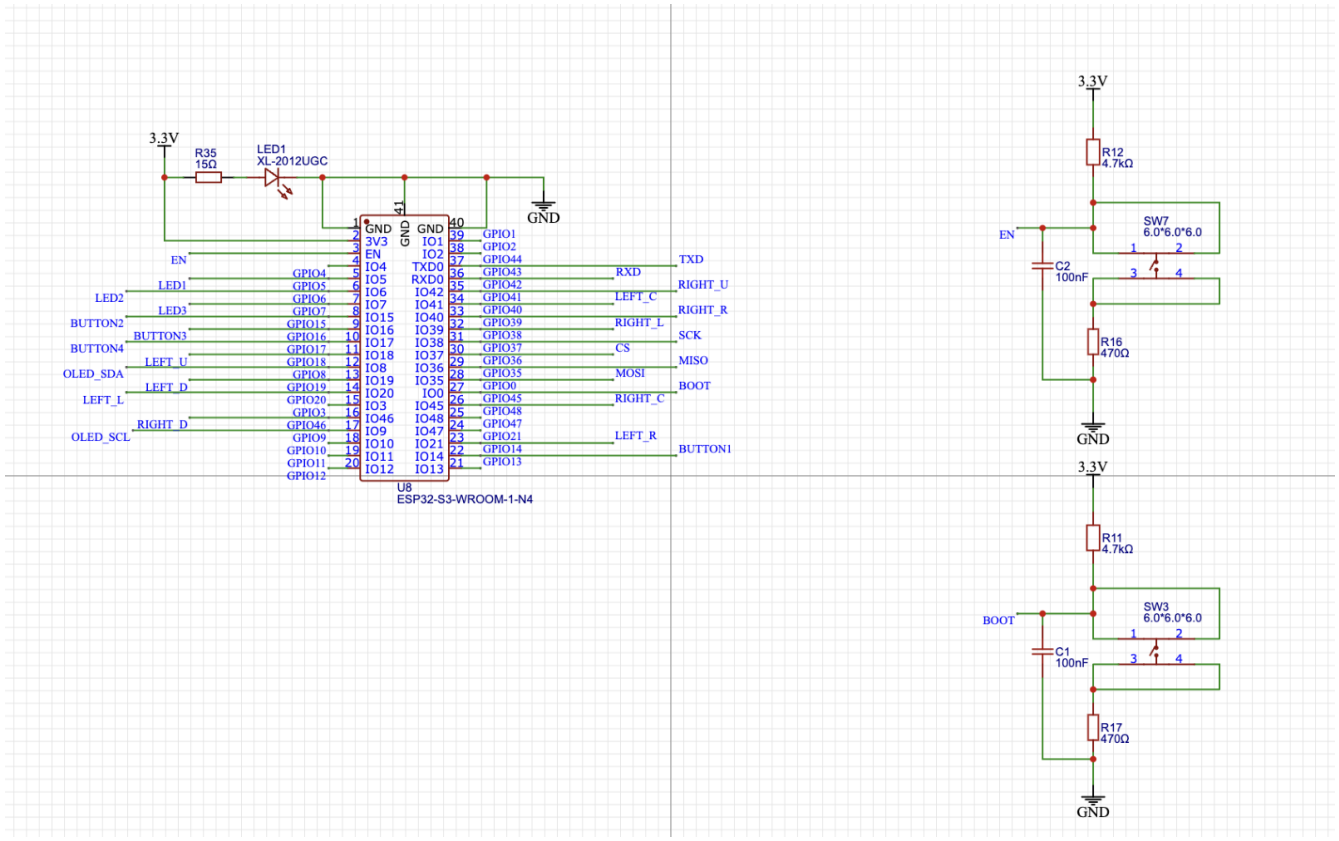
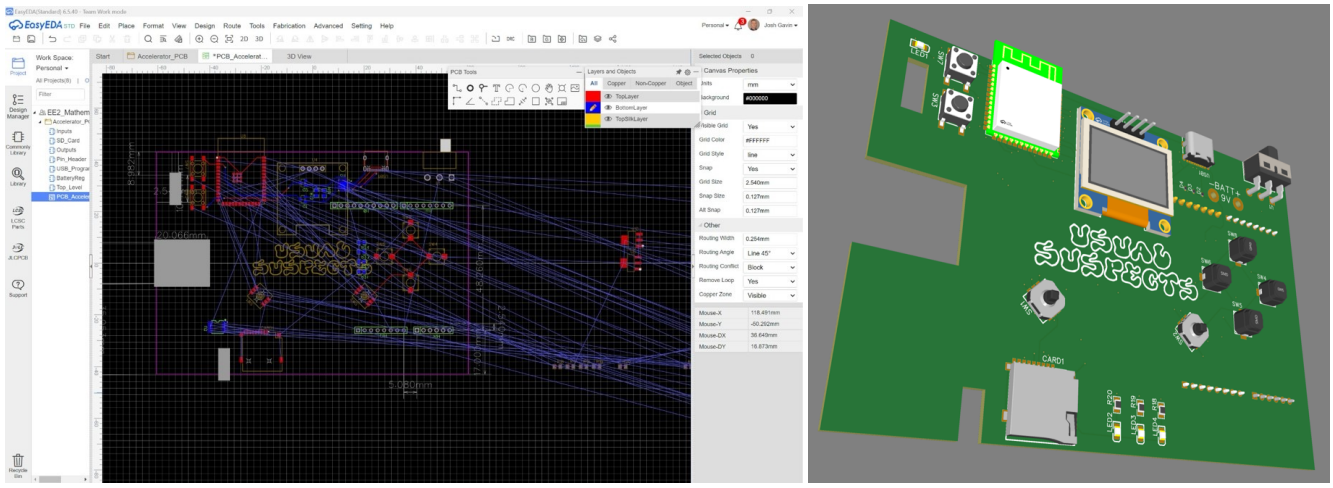


Figure 3.35: PCB Top Level Design



(a) PCB Layout with Ratlines

(b) 3D Final Model of PCB

Figure 3.36: Conversion of Schematics to PCB

Software

4.1 Website / Front-end

4.1.1 Overview

After the PCB became unfeasible due to prohibitive cost, a viable alternative was needed for the user interface. A website was the final decision after concluding it satisfied the task requirements as a low-cost, easy-to-distribute and accessible platform. Additionally the ease of bidirectional communication with the database was an appealing feature of using a website. Tab 4.1 shows a list of configurable parameters to control the Mandelbrot fractal from the website.

Parameter	**Description**
x-Offset	Horizontal Cartesian translation of the viewport center from (0,0)
y-Offset	Vertical Cartesian translation of the viewport center from (0,0)
Zoom	Linear coefficient of pixels-per-Cartesian-unit, where 1 is 100 pixels-per-unit
Max Iterations	Constant to limit engine iterations, where coordinates exceeding the maximum do not diverge (by approximation)

Table 4.1: Configurable Parameters

4.1.2 General Construction

The React.js framework was used to facilitate multiple web elements and maximise UI responsiveness. This allowed for quick addition of repeating elements, such as the parameter value input boxes, and construction of complex elements like the database communication components. After adding functional elements, CSS was employed to create a dynamic layout and styling.

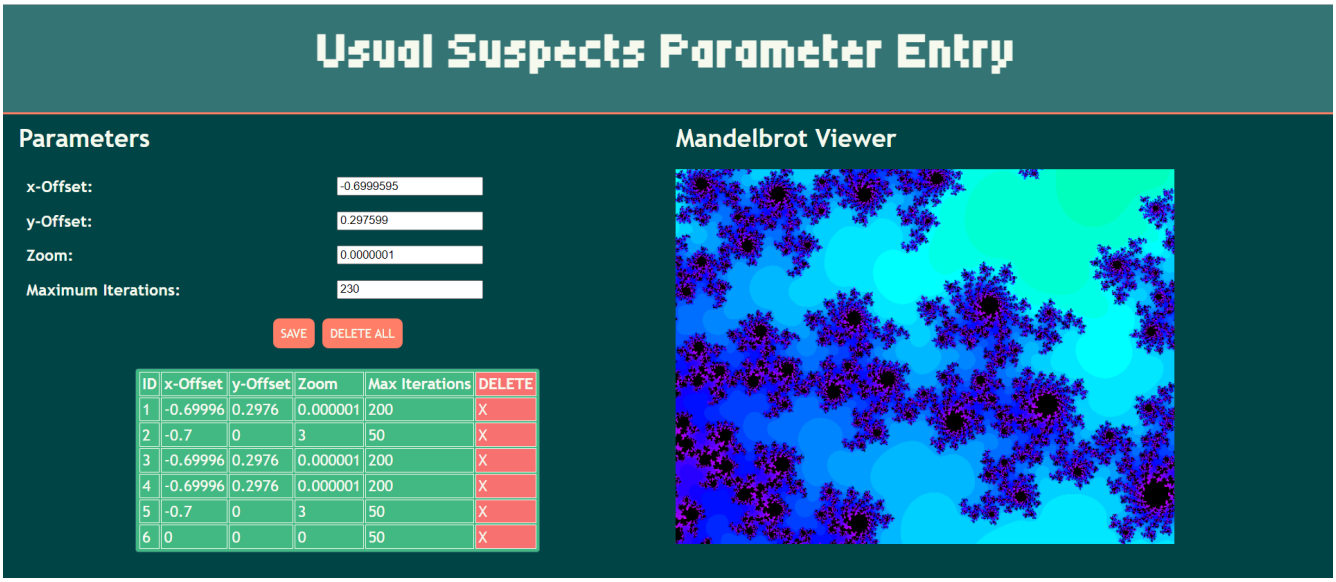


Figure 4.1: Screenshot of the Complete Website

4.1.3 Foundational Elements

Through the use of repeating React.js components, the initial website was produced swiftly which provided time for later refinement. The fundamental functionality requires inputs for parameters, so the initial components constructed were the 'NumberInput' boxes, allowing for a HTML input with labels and increment arrows. Following this, buttons were added to save the current parameter set to the database and to delete all entries. Subsection 4.1.6 gives more insight into the database communication.

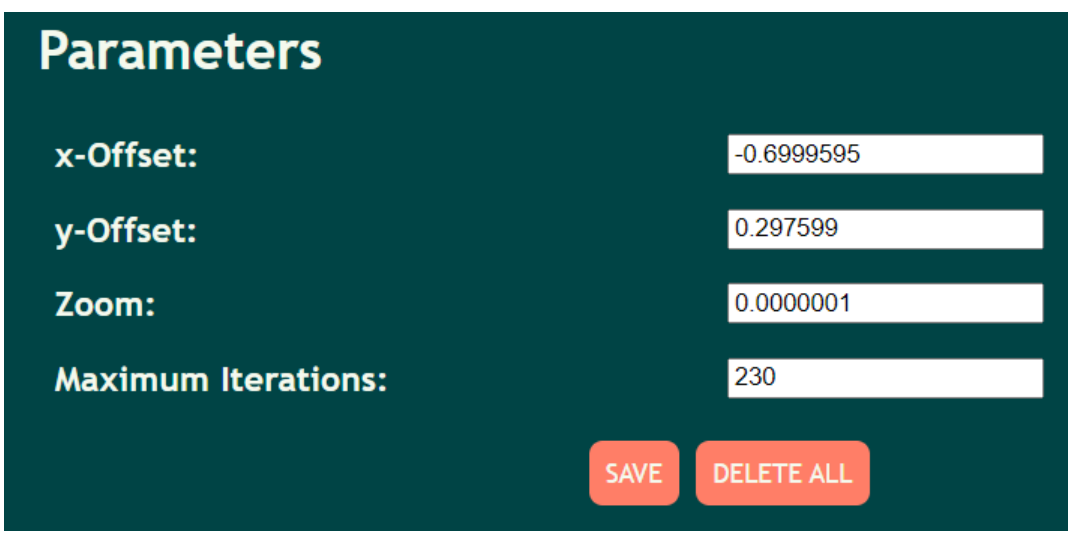


Figure 4.2: Close-up of the Website Parameter Entry Area

4.1.4 Mandelbrot Viewport Functionality

A real-time Mandelbrot fractal generator was added so that parameter changes could be easily visualised before forwarding them to the FPGA. The implementation used JavaScript to calculate the iterations of each pixel, assign a colour and display on a HTML canvas. This follows the same principles as the FPGA implementation. Using JavaScript allows for fast client-side processing with no bogging on the server. From the time of a parameter change, the image regenerates within $\tilde{0}.5$s, allowing for quick adjustments to parameters to achieve specific images. Fig. 4.3 shows a screenshot of the viewport calculating 230 iterations at 10^7 zoom, calculated in ≈ 0.3s, demonstrating the speed of the client-side calculation.

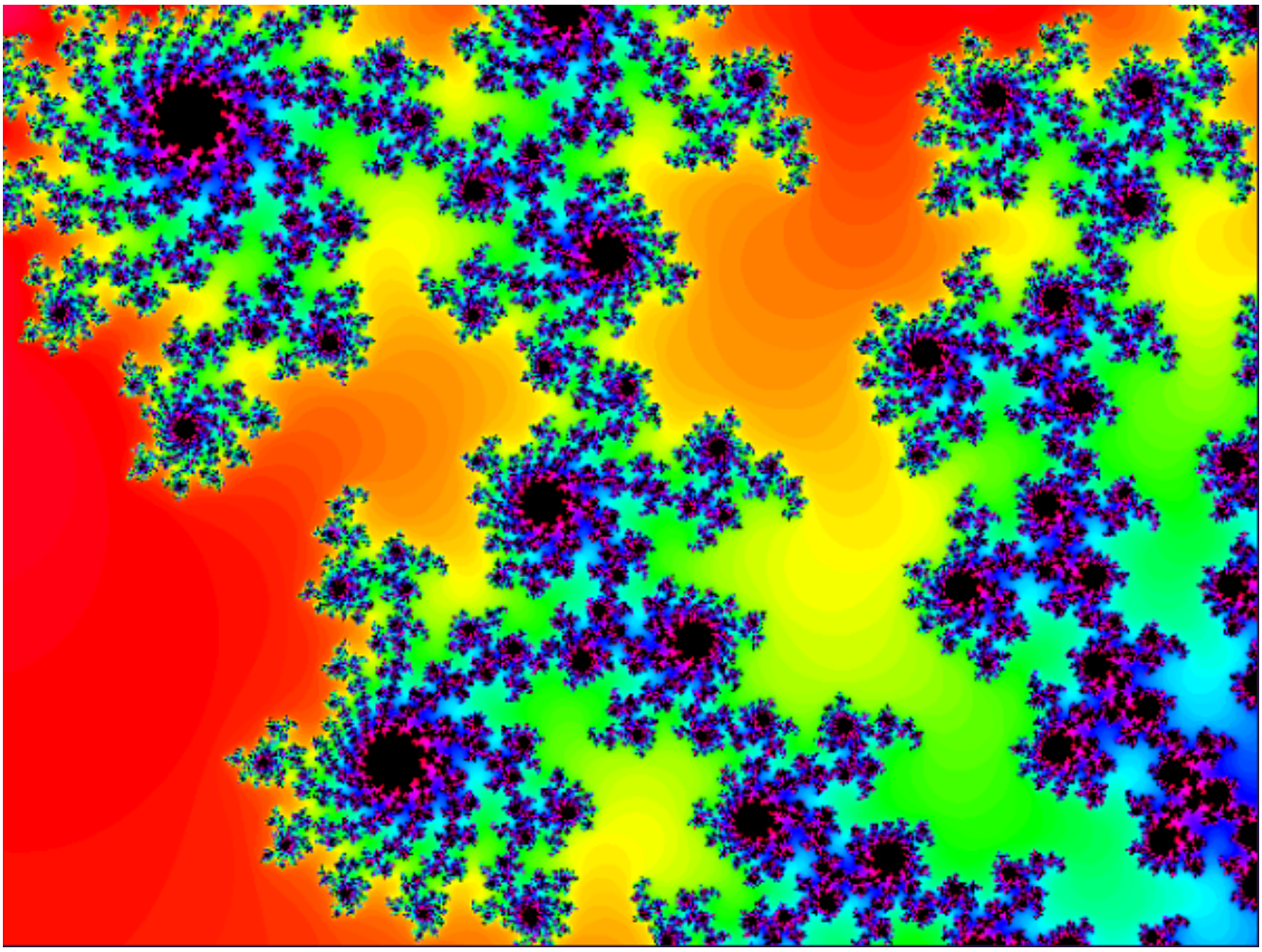


Figure 4.3: A Screenshot of the Web-client Mandelbrot Fractal

4.1.5 Mandelbrot Rendered Construction

The Mandelbrot image is rendered to a HTML canvas. Colours are defined within an array which represents up to 256 different iteration values using unique colours. The range of colour follows the continuous visible spectrum from red to violet. A `get_colour` function selects a colour from the table to represent a pixel given the number of iterations it took to diverge, or black otherwise.

```
7       const default_colors = [
22        "#AF00FF", "#B800FF", "#C100FF", "#CA00FF", "#D300FF", "#DC00FF", "#E500FF", "#EE00FF", "#F700FF",
23        "#FF00F2", "#FF00E9", "#FF00E0", "#FF00D7", "#FF00CE", "#FF00C5", "#FF00BC", "#FF00B3", "#FF00AA",
24        "#FF0098", "#FF008F", "#FF0086", "#FF007D", "#FF0074", "#FF006B", "#FF0062", "#FF0059", "#FF0050",
25        "#FF003E", "#FF0035", "#FF002C", "#FF0023", "#FF001A", "#FF0011", "#FF0008", "#FF0000", "#FF0000",
26        "#FF0700", "#FF0E00", "#FF1500", "#FF1C00", "#FF2300", "#FF2A00", "#FF3100", "#FF3800", "#FF3F00",
27        "#FF4D00", "#FF5400", "#FF5B00", "#FF6200", "#FF6900", "#FF7000", "#FF7700", "#FF7E00", "#FF8500",
28        "#FF9300", "#FF9A00", "#FFA100", "#FFA800", "#FFAF00", "#FFB600", "#FFBD00", "#FFC400", "#FFCB00",
29        "#FFD900", "#FFE000", "#FFE700", "#FFEE00", "#FFF500", "#FFFC00", "#F5FF00", "#ECFF00", "#E3FF00",
30        "#D1FF00", "#C8FF00", "#BFFF00", "#B6FF00", "#ADFF00", "#A4FF00", "#9BFF00", "#92FF00", "#89FF00",
31        "#77FF00", "#6EFF00", "#65FF00", "#5CFF00", "#53FF00", "#4AFF00", "#41FF00", "#38FF00", "#2FFF00",
32        "#1DFF00", "#14FF00", "#0BFF00", "#00FF05", "#00FF0E", "#00FF17", "#00FF20", "#00FF29", "#00FF32",
33        "#00FF44", "#00FF4D", "#00FF56", "#00FF5F", "#00FF68", "#00FF71", "#00FF7A", "#00FF83", "#00FF8C",
34        "#00FF9E", "#00FFA7", "#00FFB0", "#00FFB9", "#00FFC2", "#00FFCB", "#00FFD4", "#00FFDD", "#00FFE6",
35        "#00FFF8", "#00FFFF", "#00F6FF", "#00EDFF", "#00E4FF", "#00DBFF", "#00D2FF", "#00C9FF", "#00C0FF",
36        "#00AEFF", "#00A5FF", "#009CFF", "#0093FF", "#008AFF", "#0081FF", "#0078FF", "#006FFF", "#0066FF",
37        "#0054FF", "#004BFF", "#0042FF", "#0039FF", "#0030FF", "#0027FF", "#001EFF", "#0015FF", "#000CFF",
38        "#0400FF", "#0D00FF", "#1600FF", "#1F00FF", "#2800FF", "#3100FF", "#3A00FF", "#4300FF", "#4C00FF",
39        "#5E00FF", "#6700FF", "#7000FF", "#7900FF", "#8200FF", "#8B00FF", "#9400FF", "#9D00FF", "#A600FF",
40        "#B800FF", "#C100FF", "#CA00FF", "#D300FF", "#DC00FF", "#E500FF", "#EE00FF", "#F700FF", "#FF00FB",
41        "#FF00E9", "#FF00E0", "#FF00D7", "#FF00CE", "#FF00C5", "#FF00BC", "#FF00B3", "#FF00AA", "#FF00A1",
42        "#FF008F", "#FF0086", "#FF007D", "#FF0074", "#FF006B", "#FF0062", "#FF0059", "#FF0050", "#FF0047",
43        "#FF0035", "#FF002C", "#FF0023", "#FF001A", "#FF0011", "#FF0008", "#000000"
44      ];
```

Figure 4.4: Declaration of Canvas Object and Look-Up Table for Pixel Colours

The `check_divergence` function returns the number of iterations it takes a given pixel to diverge. A `convergence` function will return valid providing the current iterations' values are not divergent, allowing the divergence loop to continue calculating.

```
const check_divergence = (real, imag, depth) => {
  let real_old = real;
  let imag_old = imag;
  for (let index = 0; index < depth; index++) {
    if (convergence(real_old, imag_old)) {
      let real_new = real_old ** 2 - imag_old ** 2 + real;
      let imag_new = 2 * real_old * imag_old + imag;
      real_old = real_new;
      imag_old = imag_new;
    } else {
      return index;
    }
  }
  return depth;
};
```

Figure 4.5: Function to Assess the Divergence of a Pixel

The renderer iterates through all pixels in the canvas and obtains the colour for each one, drawing it to the canvas. This function is called whenever a parameter is changed on the website.

```
const render = () => {
  const left_real = parseFloat(x) - zoom / 2;
  const top_imag = parseFloat(y) + zoom / (2 * aspect_ratio);
  for (let current_real = 0; current_real < canv.width; current_real++) {
    for (let current_imag = 0; current_imag < canv.height; current_imag++) {
      const divergence_depth = check_divergence(
        left_real + current_real * zoom / canv.width,
        top_imag - current_imag * zoom / (aspect_ratio * canv.height),
        maxItr
      );
      ctx.fillStyle = get_colour(divergence_depth);
      ctx.fillRect(current_real, current_imag, 1, 1);
    }
  }
};
```

Figure 4.6: The Return Function of the Mandelbrot Renderer

4.1.6 Database Interactivity

The database graphical interface allows for the user to browse the parameter sets stored in the database. Clicking a row in the browser will load the parameter set into the input boxes, displaying an appropriate client-side preview and forwarding the relevant parameters to the FPGA. Following this functionality, a column was added to allow for the deletion of specific parameter sets in the database. Finally, CSS was used to make the layout of the table intuitive and appealing.

ID	x-Offset	y-Offset	Zoom	Max Iterations	DELETE
1	-0.69996	0.2976	0.000001	200	X
2	-0.7	0	3	50	X
3	-0.69996	0.2976	0.000001	200	X
4	-0.69996	0.2976	0.000001	200	X
5	-0.7	0	3	50	X
6	0	0	0	50	X

Figure 4.7: Image of the Embedded Database Browser

4.2 Database / Back-end

4.2.1 Overview

The frontend website and FPGA communicate with an AWS EC2 instance, which hosts a FastAPI application and SQLite database. FastAPI is a Python Web API that allows for clients to perform CRUD (Create, Retrieve, Update, Delete) operations on the SQLite database via HTTP requests (e.g. GET, POST, DELETE). The client program sends HTTP requests to different URL paths, which will call a python function based on the URL path and HTTP request in the server side python script.

```python
@app.post("/parameters")
async def create_parameters(
    x_offset: float, y_offset: float, zoom: float, max_iterations: int
):
    try:
        new_param_id = db.add_parameters(x_offset, y_offset, zoom, max_iterations)
        return {**SUCC_MESS, "parameter_id": new_param_id}
    except:
        return {**FAIL_MESS, "parameter_id": None}
```

Figure 4.8: Python Function to Add User Parameters to the Database

4.2.2 Database Design

The attribute names and types are specified in 4.2. Only a single table to store the parameters will be required for the system, therefore SQLite would be an appropriate DBMS, since it is lightweight [7] with fast read/write speeds (suitable for polling).

Attribute	Description
parameter_id	A surrogate integer key to uniquely identify each record
x_offset	A float value allowing the user to control horizontal offset.
y_offset	A float value allowing the user to control vertical offset
zoom	An integer value which allows the user to specify zoom mode

Table 4.2: Table Showing the Table Attributes and Their Types

This table is created in the database by executing the statement shown in 4.9.

```sql
CREATE TABLE IF NOT EXISTS parameters (
  parameter_id INTEGER PRIMARY KEY, x_offset REAL,
  y_offset REAL, zoom INTEGER, max_iterations INTEGER
)
```

Figure 4.9: The SQL CREATE Statement Executed to Create the Parameters Table

4.2.3 Endpoints

The functionality of each endpoint and HTTP request is defined before the backend is implemented with FastAPI and Python.

Endpoint	**Description**
/parameters/{parameter_id}	Returns the corresponding parameters in JSON format
/parameters	Returns the parameters for the current visualisation (GET). Adds parameters (inputted by the user) into the database and sends the new parameters to the Pynq board (POST)
/parameter_id	Returns all parameter ids in the database
/parameter_id/{parameter_id}	Prompts the server to send the parameters corresponding to the parameter id to the FPGA (POST). Deletes the parameters corresponding to this parameter id (DELETE)

Table 4.3: Table Showing Each URL Path and Their Roles

Each python function will execute SQL code to perform CRUD operations on the SQLite database. In order to add parameters to the database, the `INSERT OR IGNORE` statement is used. Since the parameter id is of type `INTEGER PRIMARY KEY`, a surrogate key will automatically be generated using SQL AUTO INCREMENT when a unique record of parameters is inserted into the database.

```
INSERT
OR IGNORE INTO parameters (
  x_offset, y_offset, zoom, max_iterations
)
VALUES
  (
    x_offset, y_offset, zoom, max_iterations
  )
```

Figure 4.10: The SQL INSERT Statement Executed to Insert New Parameters into the Database

The FPGA will poll the selected parameters from the database. The server will store the parameter id corresponding to the the selected parameters and execute an SQL query to return the selected parameters. The selected parameter id will be passed into the python function shown in 4.11, which will return the selected parameters.

```python
def get_parameters_by_id(self, parameter_id):
    self.c.execute(
        """SELECT x_offset, y_offset, zoom, max_iterations
                    FROM parameters
                    WHERE parameter_id = ?""",
        (parameter_id,),
    )
    return self.c.fetchone()
```

Figure 4.11: The SQL SELECT Statement to Query the Database for Parameters

The user application allows for parameters to be deleted from the table, so the server should include a python function to implement this functionality. The python function in 4.12 accounts for the edge case where the parameters that the user is trying to delete are the selected parameters which are being polled by the FPGA, as the selected parameter id is updated to the last parameter id in the table in this case.

```python
def delete_parameters(self, parameter_id):
    self.c.execute("""DELETE FROM parameters WHERE parameter_id =
        ?""", (parameter_id,))
    if parameter_id == self.__select_parameter_id:
        self.__select_parameter_id = self.c.lastrowid
    self.conn.commit()
```

Figure 4.12: The SQL DELETE Statement Used to Delete a Record in the Table

4.3 Embedded Software

4.3.1 FPGA-Side Python Client

The FPGA-side embedded software had to interface with multiple components: the database to receive parameters, the accelerator synthesised on the FPGA and the output to HDMI. This could've been implemented using multiple programs, however the decision was made to have a single python script handle all the previously defined processes. This would become beneficial in testing the performance of the accelerator as both of its endpoints were handled by a single program.

In order to implement a program which could handle multiple interfaces at once, a concurrent approach was used. This meant making use of the 'asyncio' python library. This was decided over a multi-threaded approach, as it provides greater consistency in program execution and output, making debugging much easier.

Three async functions were defined, as shown in Fig. 4.13-4.16. One function was responsible

for polling the server for new parameters every few seconds. Another fed parameters into the FPGA and the final was responsible for outputting the image via HDMI.

```
async def poll():
    while True:
        try:
            response = requests.get(URL, verify=False)
            if response.status_code == 200: # if request succeded
                data = response.json()
                x = int(data['x_offset'])
                y = int(data['y_offset'])
                zoom = int(data['zoom'])
                maxitr = int(data['max_iterations'])

                await update_hw(x, y, zoom, maxitr)
            else: # in case of failed request
                print(f"HTTP Request Failed")
                print(f"Status code: {response.status_code}")
        except Exception as e:
            print(f"Exception occured: {e}")
        await asyncio.sleep(POLL_INTERVAL)
```

Figure 4.13: Async Poll Function

```
async def update_hw(x, y, zoom, maxitr):
    await send_to_fpga(x, y, zoom, maxitr)
    await hdmi_output()
```

Figure 4.14: Async Update Hardware Function

The first and only function called from main is the `poll` function defined in Fig. 4.13. This function defines the main continuous while loop for the program. At each iteration of this loop the program requests the most recent parameter added to the database and assigns the received data to variables. It then awaits the completion of the async function `update_hw`. The function `update_hw` is a simple intermediary function to ensure that data is sent to the FPGA, before being outputted via HDMI.

```
async def send_to_fpga(x, y, zoom, maxitr):
    pixgen = overlay.pixel_generator_0

    pixgen.register_map.gp0 = x
    pixgen.register_map.gp1 = y
    pixgen.register_map.gp2 = zoom
    pixgen.register_map.gp3 = maxitr
```

Figure 4.15: Async Send to FPGA Function

The FPGA interface is handled in the `send_to_fpga` async function (see Fig. 4.15). Using inbuilt PYNQ modules the program can interface with the synthesised design via Memory-Mapped I/O (MMIO). The values for the x offset, y offset, zoom and maximum iterations are assigned to registers in the register file. In this case x is assigned to `regfile[0]`. Once these values have been written, the program returns to the `update_hw` function and then to the `hdmi_output` function (see Fig. 4.16).

```
async def hdmi_output():
    frame = imgen_vdma.readframe()
    hdmi_out.writeframe(frame)
```

Figure 4.16: Async Output HDMI Function

This function simply reads frames into a cycle of memory buffers and writes it to the HDMI output. All these interfaces and modules were setup at the beginning as shown in Fig. 4.17.

```
# SERVER SIDE
URL = "http://ec2-18-130-114-167.eu-west-2.compute.amazonaws.com:8080/parameters/"
POLL_INTERVAL = 5 # Interval for polling of server


# HARDWARE SIDE
X_RES = 640
Y_RES = 480
overlay = Overlay("/home/xilinx/jupyter_notebooks/base_wrapper.bit")
imgen_vdma = overlay.video.axi_vdma_0.readchannel
videoMode = common.VideoMode(640, 480, 24)
imgen_vdma.mode = videoMode
imgen_vdma.start()
hdmi_out = overlay.video.hdmi_out
hdmi_out._vdma = overlay.video.axi_vdma
hdmi_out.configure(videoMode)
hdmi_out.start()
```

Figure 4.17: FPGA Client Program Setup

4.3.2 ESP32 GUI / Simulation

The ESP32 GUI was developed for the aforementioned initial design. It was designed entirely via Wokwi [8], a simulator providing an environment for IoT project development. In the initial design, the ESP32 was responsible for handling user input, housing two five-way tactile switches and four buttons. The GUI was implemented using MicroPython and provided a number of features including: parameter handling via the five-way tactile switches, a menu for selecting and browsing parameters stored on the database and a menu for assigning specific values for parameters. The developer view of the simulation can be seen in Fig. 4.18.

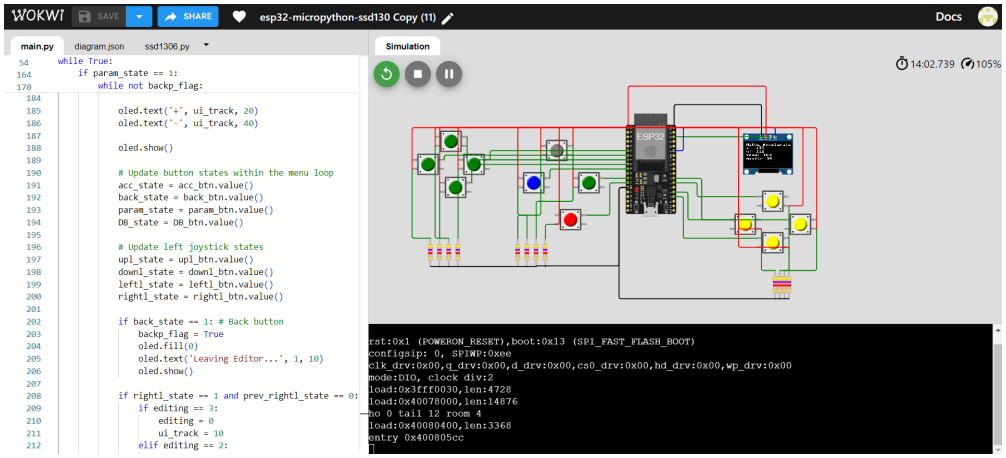
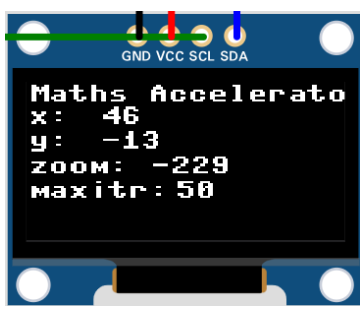


Figure 4.18: ESP32 Wokwi Simulation Development View

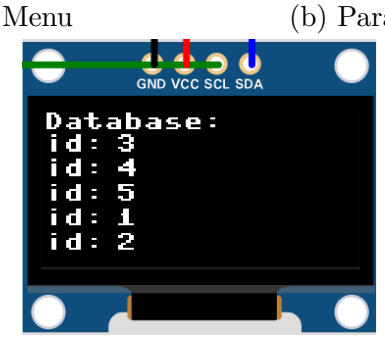
The simulation included twelve buttons, eight to model the two five-way tactile switches and the rest for the other buttons. They were each wired as defined in the PCB design, using a $4.7k\Omega$ pull-down resistor (see Fig. 3.29). Similarly the OLED screen was assigned to the appropriate pins on the ESP32.

The navigation between menus was implemented using nested while loops with conditional statements defining how the system interacts with user inputs. Each menu has an exit state which can be satisfied by the user. The parameter, database and main menu can be seen in Fig. 4.19.



(a) Main Menu

(b) Parameter Menu



(c) Database Menu

Figure 4.19: ESP32 GUI Menus

```
while not backp_flag:
    #SETUP PARAM EDITOR UI
    oled.fill(0)
    oled.text('Param Editor:', 1, 1)

    oled.text('x', 10, 10)
    oled.text('y', 35, 10)
    oled.text('zoom', 55, 10)
    oled.text('itr', 100, 10)

    oled.text(str(img[0]), 10, 30)
    oled.text(str(img[1]), 35, 30)
    oled.text(str(img[2]), 55, 30)
    oled.text(str(img[3]), 100, 30)

    oled.text('+', ui_track, 20)
    oled.text('-', ui_track, 40)

    oled.show()
```

Figure 4.20: Code Implementation of Parameter Menu

The user parameters are stored in an array `img = [x, y, zoom, max_itr]`, allowing easy and simple access to each of them. An example of the array values being updated based on button states can be seen in Fig. 4.21.

```
# Left joystick controls
if upl_state == 1: # y+
    img[1] += 1
if downl_state == 1: # y-
    img[1] -= 1
if rightl_state == 1: # x+
    img[0] += 1
if leftl_state == 1: # x-
    img[0] -= 1

# Right joystick controls
if upr_state == 1: # fine zoom (+)
    img[2] += 1
if downr_state == 1: # fine zoom (-)
    img[2] -= 1
if rightr_state == 1: # coarse zoom (+)
    img[2] += 5
if leftr_state == 1: # coarse zoom (-)
    img[2] -= 5
```

Figure 4.21: Code Implementation of Joystick Control

Ultimately, due to the reasons discussed in the planning and design section, the ESP32 was eventually scrapped during development. The website was then used for the UI.

Evaluation and Testing

5.1 Overview

This section looks to evaluate whether the final system implementation accurately and appropriately satisfies the defined requirements set out in Tab. 2.4. This includes a discussion and thorough walk-through of the component and requirement testing process. Additionally, this section provides an analysis on the final system and its sub-components, describing where things could've gone better.

5.2 Hardware

5.2.1 Software Implementation

As discussed in Tab. 2.4 the performance benchmark for the hardware implementation was defined by a software implementation of the function visualisation. Two benchmarks were developed, one in Python and another in Cython. The benchmarks averaged the time taken to calculate the Mandelbrot Set for varying 'maximum iterations' of the function's inner loop (ranging from 50-500) and generating a 640x480 image. The results of this benchmark on a 13th Gen Intel Core i9-13900H can be seen in Fig. 5.1.

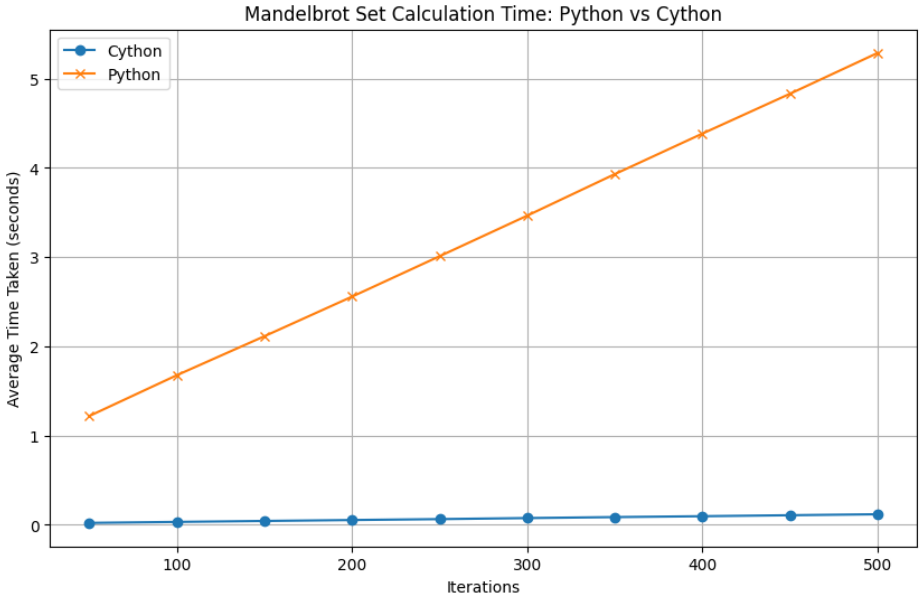


Figure 5.1: Cython vs Python Software Benchmark

The Python implementation was relatively simple, requiring only three functions. A snippet of the code can be seen in Fig. 5.2. The `mandelbrot` function is responsible for performing the Mandelbrot calculation $z^2 + c$ and determines divergence based off the `max_iter` parameter. The `mandelbrot_set` function performs the Mandelbrot calculation for a defined range of values. These are equivalent to the zoom, offset and resolution parameters used in the hardware implementation. Finally the `time_mandelbrot` function calculates the time taken to calculate the entire Mandelbrot set.

```python
def mandelbrot(c, max_iter):
    z = c
    for n in range(max_iter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return max_iter

def mandelbrot_set(xmin, xmax, ymin, ymax, width, height, max_iter):
    r1 = np.linspace(xmin, xmax, width)
    r2 = np.linspace(ymin, ymax, height)
    n3 = np.empty((width, height))
    for i in range(width):
        for j in range(height):
            n3[i, j] = mandelbrot(r1[i] + 1j*r2[j], max_iter)
    return (r1, r2, n3)

def time_mandelbrot(xmin, xmax, ymin, ymax, width, height, max_iter):
    start_time = time.time()
    mandelbrot_set(xmin, xmax, ymin, ymax, width, height, max_iter)
    end_time = time.time()
    return end_time - start_time
```

Figure 5.2: Python Benchmark Code Snippet

The Cython implementation had different syntax, but was functionally the same. The equivalent functions seen in Fig. 5.2 for the Python implementation, can be seen in Fig. 5.3 for the Cython implementation.

```cython
def mandelbrot(complex c, int max_iter):
    cdef complex z = c
    cdef int n
    for n in range(max_iter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return max_iter

def mandelbrot_set(double xmin, double xmax, double ymin, double ymax, int width, int height, int max_iter):
    cdef np.ndarray[np.float64_t, ndim=1] r1 = np.linspace(xmin, xmax, width)
    cdef np.ndarray[np.float64_t, ndim=1] r2 = np.linspace(ymin, ymax, height)
    cdef np.ndarray[np.int32_t, ndim=2] n3 = np.empty((width, height), dtype=np.int32)
    cdef int i, j
    for i in range(width):
        for j in range(height):
            n3[i, j] = mandelbrot(r1[i] + r2[j] * 1j, max_iter)
    return r1, r2, n3

def time_mandelbrot(double xmin, double xmax, double ymin, double ymax, int width, int height, int max_iter):
    start_time = time.time()
    mandelbrot_set(xmin, xmax, ymin, ymax, width, height, max_iter)
    end_time = time.time()
    return end_time - start_time
```

Figure 5.3: Cython Benchmark Code Snippet

Finally two more programs were developed. The first, seen in Fig. 5.4a, performed testing of

the Mandelbrot program for varying maximum iterations. It also allowed the tester to specify the input parameters and the number of runs for each maximum iteration step to establish an average. This program then saved the results in a .csv file. The other program, seen in Fig. 5.4b, reads the results from the .csv files and plots them.

```python
# Define the plot parameters
xmin, xmax, ymin, ymax = -2.0, 1.0, -1.5, 1.5
width, height = 640, 480

# Define the iteration steps you want to test
iteration_steps = [50, 100, 150, 200, 250, 300, 350, 400, 450, 500]

# Collect timing data
data = {'iterations': [], 'avg_time_taken': []}
num_runs = 50

for max_iter in iteration_steps:
    total_time = 0
    for _ in range(num_runs):
        start_time = time.time()
        mandelbrot.mandelbrot_set(xmin, xmax, ymin, ymax, width, height, max_iter)
        end_time = time.time()
        total_time += end_time - start_time
    avg_time_taken = total_time / num_runs
    data['iterations'].append(max_iter)
    data['avg_time_taken'].append(avg_time_taken)
    print(f"Iterations: {max_iter}, Average Time taken: {avg_time_taken:.4f} seconds over {num_runs} runs")

# Create a DataFrame and save to CSV
df = pd.DataFrame(data)
df.to_csv('mandelbrot_timing_cython.csv', index=False)

print("Timing data saved to 'mandelbrot_timing_cython.csv'")
```

(a) Benchmark Testing Program

```python
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV files
cython_data = pd.read_csv('Cython_Benchmark/mandelbrot_timing_cython.csv')
python_data = pd.read_csv('Python_Benchmark/mandelbrot_timing_python.csv')

# Plotting the data
plt.figure(figsize=(10, 6))

plt.plot(cython_data['iterations'], cython_data['avg_time_taken'], label='Cython', marker='o')
plt.plot(python_data['iterations'], python_data['avg_time_taken'], label='Python', marker='x')

plt.xlabel('Iterations')
plt.ylabel('Average Time Taken (seconds)')
plt.title('Mandelbrot Set Calculation Time: Python vs Cython')
plt.legend()
plt.grid(True)

plt.savefig('mandelbrot_timing_comparison.png')  # Save the plot as an image file
plt.show()
```

(b) Benchmark Analysis Program

Figure 5.4: Software Benchmark Infrastructure

Analysing the results of both methods (Fig. 5.1), the Cython implementation was used as the performance goal for computation. Therefore, in order to meet the system requirements, the hardware had to perform the Mandelbrot computation faster than the Cython implementation. A linear model was established for the Cython performance to indicate deterministically its performance. This model is captured with the results seen in Fig. 5.5 and Tab. 5.1

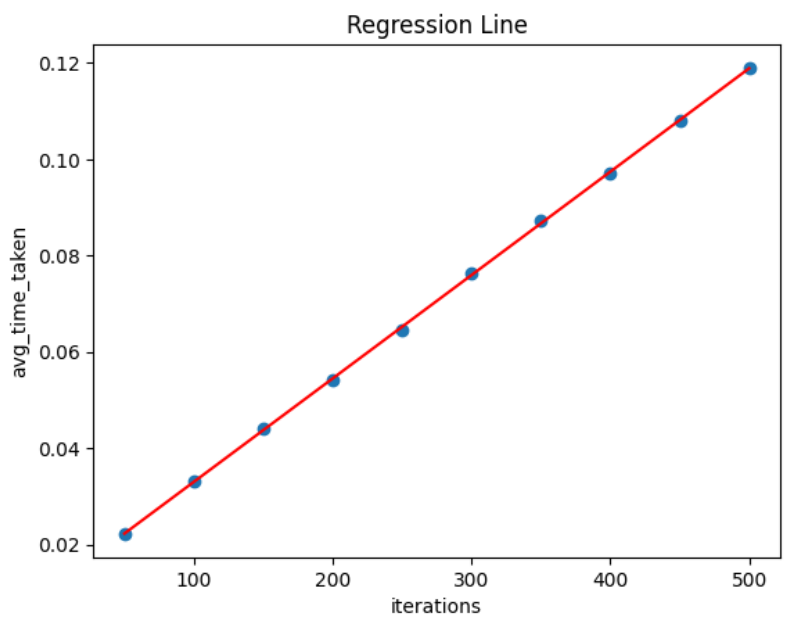


Figure 5.5: Graphical Result of Cython Linear Regression Model

Model Component	Value
Slope	0.0002147419536474979
Intercept	0.011536491711934369
R-squared	0.9998846682637785

Table 5.1: Numerical Result of Cython Linear Regression Model

Therefore ideally the final implementation would have the following attributes: A smaller slope, as this would suggest better performance due to time increasing slower with increasing iterations. Also a lower intercept, as this would indicate that the starting time is lower for a baseline iteration. Finally a smaller R-squared value, although not holistically, would suggest that iterations have less influence on computation time, which is desired. With this defined, the hardware implementation could be evaluated effectively with regards to the system requirements.

5.2.2 Hardware Comparison

In an attempt to closely emulate the conditions of the software implementation, a 640x480 image was generated on the FPGA and the time taken to generate a frame was measured and averaged. The regression model results can be seen below in Fig. 5.6 and Tab. 5.2.

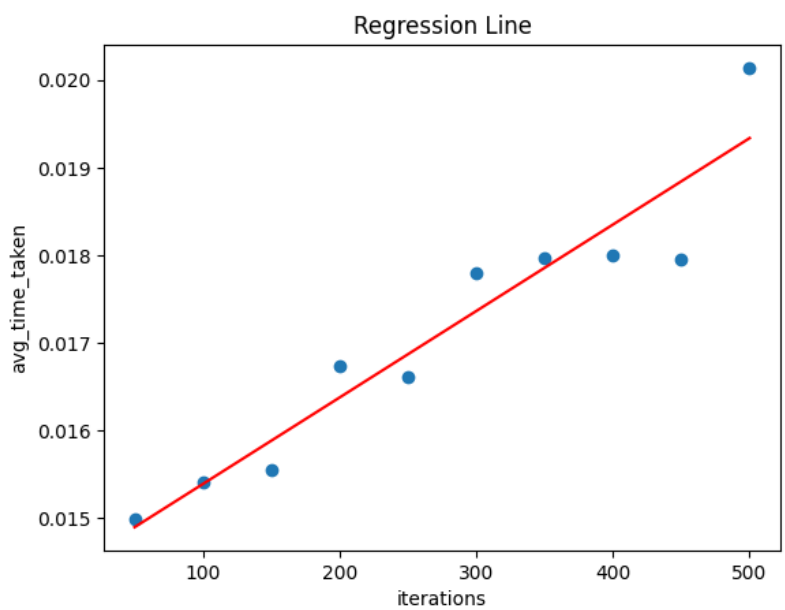


Figure 5.6: Graphical Result of Hardware Linear Regression Model

Model Component	Value
Slope	0.000009859139151515151
Intercept	0.014406829933333334
R-squared	0.90667784080308

Table 5.2: Numerical Result of Hardware Linear Regression Model

These results confirm that the requirements have been comfortably met. The gradient is far smaller, by orders of magnitude, showing that the system's hardware accelerator is able to calculate higher iteration Mandelbrot sets significantly faster than Cython. This is supported by the smaller R-squared value as well, which suggests that increasing iterations have a smaller impact on the performance of the hardware. However it's important to note that the intercept is actually larger. This suggests that the parallelised design is less beneficial for low iteration visualisations of the Mandelbrot. This can be seen in Fig. 5.7, which shows a direct comparison between the two implementations. It's clear that between 0-50 iterations there is a range where Cython performs better than the hardware.

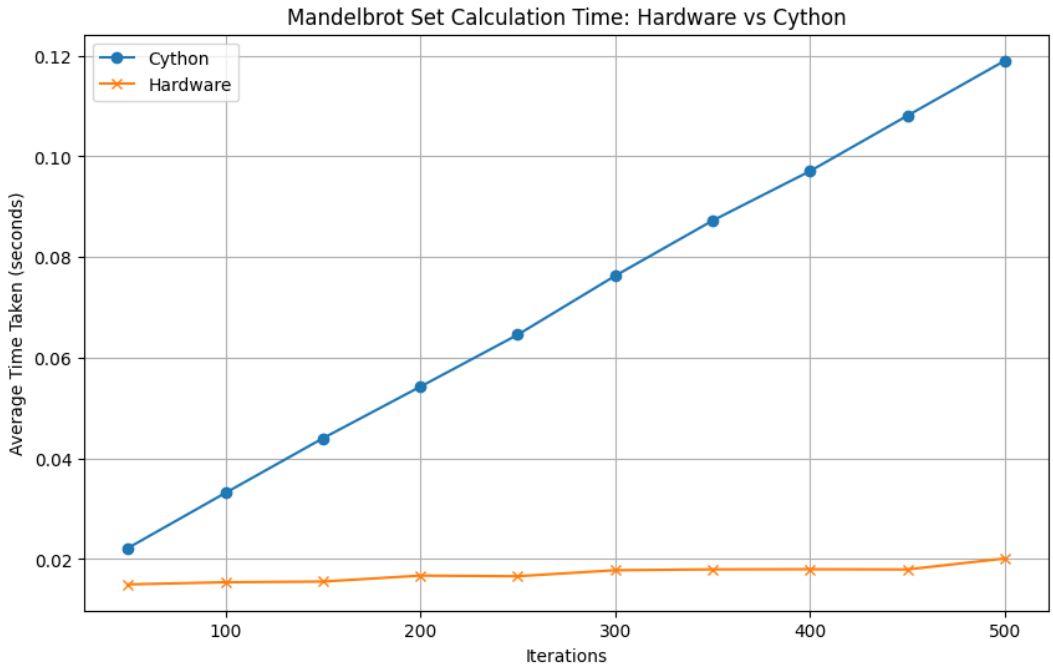


Figure 5.7: Cython vs Hardware Results

5.3 Software

5.3.1 UI - Subjective Testing

Due to the nature of the UI requirements, testing could not be performed objectively. In order to evaluate whether the final design was accessible, intuitive and effective as an educational tool, subjective testing was required. This was done using the following method: Allow a fifteen minute, in-person testing period, letting individuals try the UI for themselves. After this they were given a questionnaire, asking questions pertaining to the system requirements. The aforementioned questions have been broken down in Tab. 5.3.

Question	**Justification**
Do you have any visual impairments?	Determine whether the tester has any visual accessibility concerns
Do you have any physical disabilities?	Determine whether the tester has any physical accessibility concerns
Can you rate your experience adjusting parameters from a scale of 1-10?	Numerically gauge the ease-of-use for parameter adjustment
Can you rate your experience using the database functionality from a scale of 1-10	Numerically gauge the ease-of-use for the database functionality
On a scale from 1-10, how effective was the database functionality for "bolstering the educational experience"?	Numerically gauge the effectiveness of the database functionality as an educational feature
From a scale of 1-10, how intuitive was the design for you?	Numerically gauge the intuitiveness of the design
If you answered "Yes" to either of the first 2 questions, from a scale of 1-10, how easy was it to interface with the UI?	Numerically gauge the accessibility of the design for those who had impairments / disabilities

Table 5.3: Questionnaire Questions for Subjective UI Testing

Unfortunately due to time constraints only ten respondents were available for testing. Because of this, insufficient data was available to assess the accessibility of the UI design for individuals with visual impairments and physical disabilities. A distribution of respondents with the aforementioned accessibility concerns can be seen in Fig. 5.8. The only respondent who answered "Yes" had a minor astigmatism, which decidedly was not sufficient to provide a holistic conclusion for visual accessibility concerns.

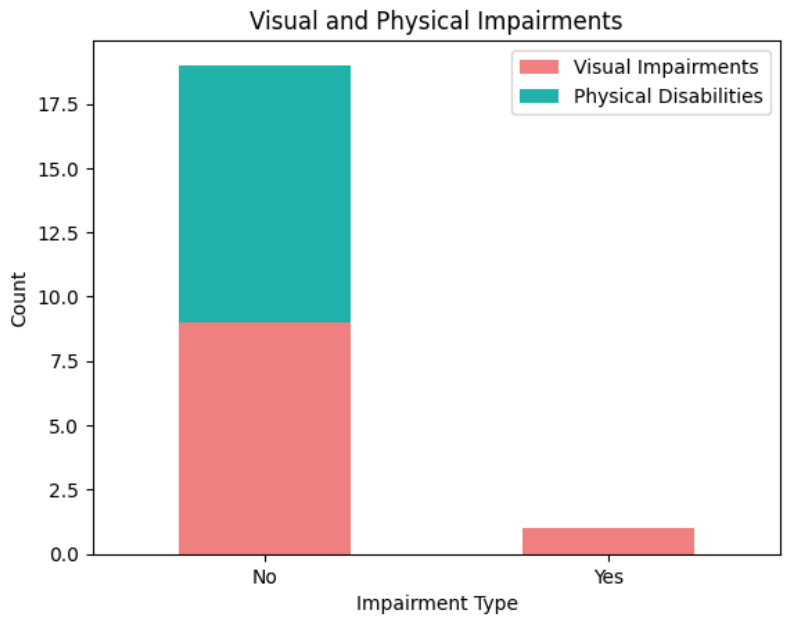
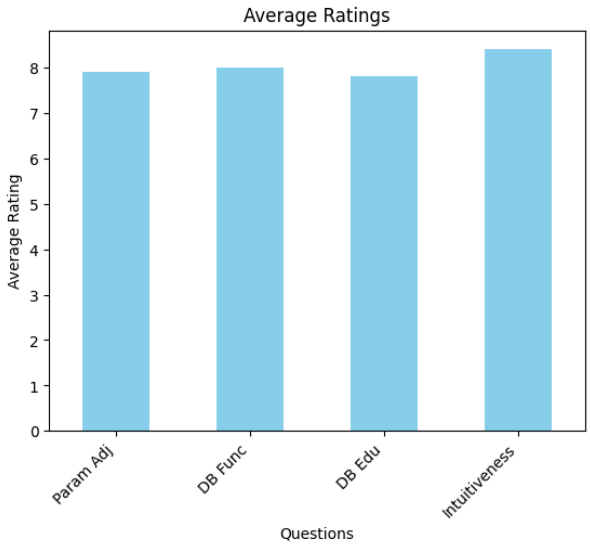
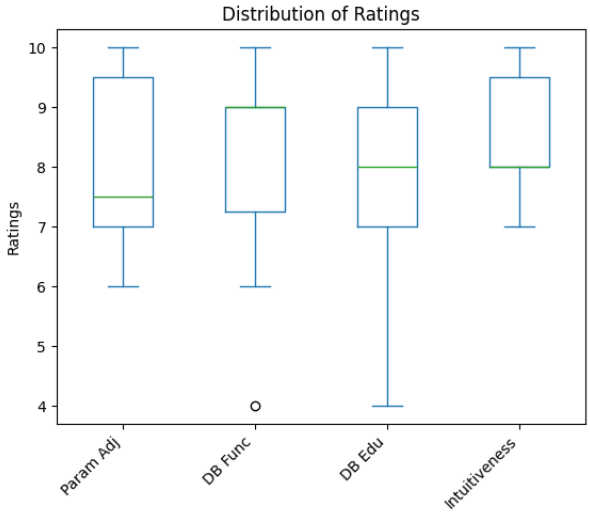


Figure 5.8: Distribution of Respondents with Visual Impairments and Physical Disabilities

The rest of the results can be seen in Fig. 5.9. The average rating provided for each of the 'rating' questions can be seen in Fig. 5.9a. These are very positive results with an overall average of 8.025. This suggests a general satisfaction across the board with the current implementation. The distribution of these results, providing a more detailed story of the data, can be seen in Fig. 5.9b. Seeing that a rating of '10' was achieved on all 'rating' questions and each question's lower quartile remained above '7', supports the conclusion made regarding Fig. 5.9a. However we also see that there was dissatisfaction with some respondents regarding the database's ability to "bolster the educational experience". This could be to do with the fact the database's appeal as an educational tool is to do with it's interactivity (sharing capabilities) with students. Given that testing was performed on an individual basis, this might've impacted how the database functionality was perceived amongst testers.



(a) Average Rating Per Question



(b) Distribution of Ratings Per Question

Figure 5.9: Subjective Testing Results

A final question at the end of the questionnaire asked testers to suggest changes to improve the current implementation of the UI. These results highlighted a few areas for improvements. The first being colour scheme with suggestions / comments such as: "Make it more visually appealing by making it look more modern... ", "Change text colours to be bolder in comparison to background" and "the green text on green background makes the text less clear". Additional aesthetic suggestions included: " softer curves and more readable fonts", "The numbers in the database are difficult to interpret" and "I would increase the size of the table relative to the inputs". Despite good results regarding intuitiveness, respondents suggested various aesthetic changes. This suggests that the design is usable and easy to understand, but not as pleasant visually as it could be. Functional improvements were also suggested: "sliders for changing the parameters", "a way to differentiate between the entries in the table in a meaningful manner (e.g. if we could name them of if they could be presented with some kind of preview)" and "implement zoom on the actual thing" (regarding the Mandelbrot viewer). These answers suggest that the user experience can be improved by introducing more functionality with regards to modifying parameters and data.

Overall it was concluded that based on these results, to the best of our knowledge, the requirements were met. However accessibility requirements still remain questionable due to the lack of respondents with visual impairments or physical disabilities. Additionally, recommended respondent suggestions should be considered to ensure the user experience is sufficient for an educational environment. This includes changing the colour scheme and aesthetics, as well as providing greater functionality and options for the users to modify data / parameters. Despite this, results suggest the UI is intuitive, allows for easy adjustment of parameters and provides database

functionality to bolster the educational experience.

5.3.2 Website Alteration from Feedback

From the results of the survey, the impact of colourblindness on the usability of the website was considered. Deuteronomy, a green-weak variant, is the most common form of colourblindness. This means the original design of the website would not be perceived as easily by a colourblind user, due to the use of a green palette. No colourblind survey members were available to give live feedback, so an online Deuteronomy filter [9] was used to assess changes in website design. From the comparison in Fig. 5.11, the purple website revision can be seen to significantly improve the readability of the page by reducing the monochromatic nature of the web elements.

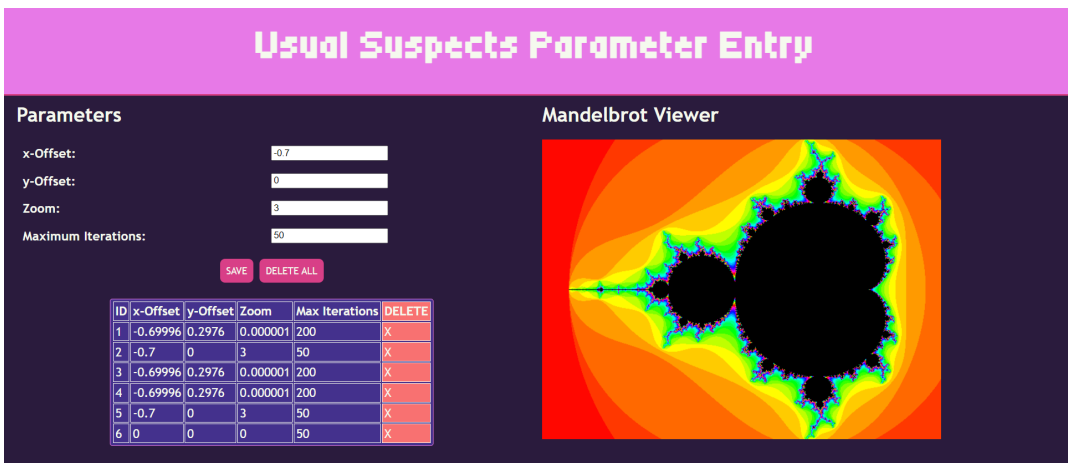
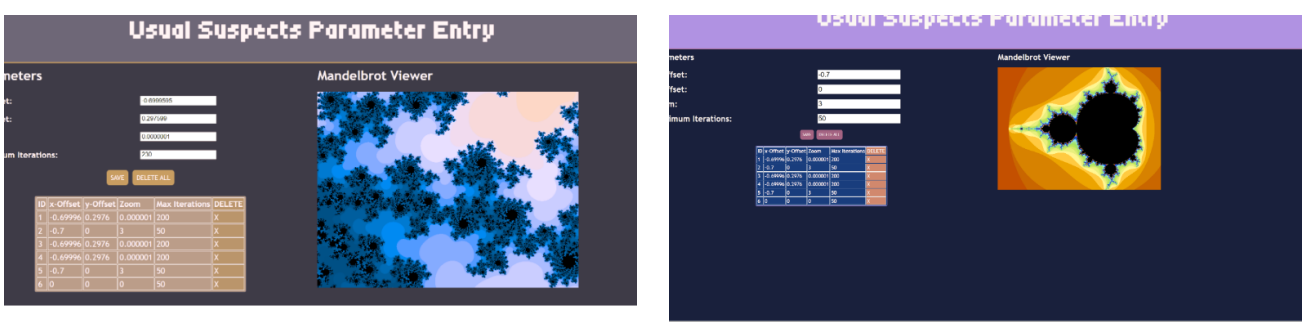


Figure 5.10: Amended Website Design Based on Suggestions



(a) Green Website Through Deuteronomy Filter (b) Purple Website Through Deuteronomy Filter

Figure 5.11: Comparison of Old and New Website Through Deuteronomy Filter

5.3.3 Connection / Transmission Performance

The core communication within the system is with the server. As mentioned in the project requirements (Tab. 2.4), fast and effective transmission is necessary. The relevant connections to test against this requirement were between the FPGA and the server, and between the front-end client and the server. The Round-Trip Time (RTT), its standard deviation and jitter for both these connections were calculated using the code in Fig. 5.12.

```python
def calculate_rtt_and_jitter(url, count):
    # store each run's RTT
    rtts = []

    # make count number of requests to the URL and measure time taken
    for _ in range(count):
        start_time = time.time()
        try:
            response = requests.get(url)
            end_time = time.time()
            if response.status_code == 200:
                rtt = end_time - start_time
                rtts.append(rtt)
        except requests.RequestException:
            pass

    # calculate average, standard deviation of RTTs, and jitter
    if len(rtts) > 0:
        avg_rtt = statistics.mean(rtts)
        stddev_rtt = statistics.stdev(rtts)
        if len(rtts) > 1:
            jitter = statistics.mean([abs(rtts[i] - rtts[i - 1]) for i in range(1, len(rtts))])
        else:
            jitter = 0.0
        return avg_rtt, stddev_rtt, jitter
    else:
        return None, None, None

if __name__ == "__main__":
    count = 500
    url = "http://ec2-18-130-114-167.eu-west-2.compute.amazonaws.com:8080"
    avg_rtt, stddev_rtt, jitter = calculate_rtt_and_jitter(url, count)

    if avg_rtt is not None:
        print(f"Average RTT: {avg_rtt:.4f} seconds")
        print(f"Standard Deviation of RTT: {stddev_rtt:.4f} seconds")
        print(f"Jitter: {jitter:.4f} seconds")
    else:
        print("Failed to calculate RTTs and jitter. No successful requests.")
```

Figure 5.12: Python Code for Calculating Connection Performance Metrics Between End-Point and Server [10]

The results of this test can be seen in Tab. 5.4 for a sample of five hundred requests. Despite increased variability and decreased stability in the Web-Client's performance, it still performs well overall and is sufficient for our use-case. These characteristics can be seen in the relatively high jitter and standard deviation of RTT. The FPGA performed very well and comfortably meets the requirements for the system.

End-Point	**Avg RTT (ms)**	**σ of RTT (ms)**	**Jitter (ms)**	**Packet Loss (%)**
Web-Client	35.6	105.2	26.9	0
FPGA	36.8	20.7	6.0	0

Table 5.4: Connection Performance Metrics

With these connections accounted for, the requirement for reliable and seamless transmission can be considered satisfied. Ultimately this requirement was designed to ensure that user-perceived latency is minimised, which can can now be confirmed by these metrics.

Conclusion

6.0.1 Overview

Overall the final system was a success, satisfying the defined requirements to a good extent. Unfortunately due to time constraints the testing was not entirely holistic, with regards to the website design, however improvements were made to sufficiently answer concerns from subjective testing. The hardware results were extremely positive, given that amount of time invested in designing, developing and debugging. The final system provides students with high quality visualisations of the Mandelbrot set, allowing a good range of manipulations for them to learn from and have fun.

6.0.2 Future Improvements

Given more time, these are the changes / improvements that would be made to the system:

- Addition of functionality to the website - This includes more ways to change parameters, interact with the Mandelbrot viewer and more educational elements. These educational elements could include, equations and walk-throughs / explanations about the Mandelbrot set.
- Another useful improvement could be expanding on the database / server-side functionality. This could be allowing for profiles, to share images or some kind of teacher-student communication system.
- Hardware improvement could be made regarding the `packer.v` file. At the moment it only takes one pixel at a time per handshake from the hardware design. This could be improved by allowing a row of pixels to be processed at once. In theory this would greatly improve performance.

Bibliography

[1] E. Stott, "Accelerator guide," June 2024. [Online]. Available: https://github.com/edstott/EE2Project/blob/main/doc/accelerator-guide.md

[2] TeamGantt, "Gantt chart creation software," May 2024. [Online]. Available: https://www.teamgantt.com/

[3] V. Verify, "Synchronous fifo," December 2022. [Online]. Available: https://vlsiverify.com/verilog/verilog-codes/synchronous-fifo/

[4] V. Adams, "Implementing the mandelbrot set on de1-soc," May 2024. [Online]. Available: https://vanhunteradams.com/DE1/Mandelbrot/Mandelbrot_Implementation.html#Implementing-this-arithmetic-in-Verilog

[5] *PYNQ-Z1 Reference Manual*, 2023. [Online]. Available: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual

[6] EasyEDA, "Free pcb design software," May 2024. [Online]. Available: https://easyeda.com/

[7] javatpoint, "Sqlite advantages," May 2024. [Online]. Available: https://www.javatpoint.com/sqlite-advantages-and-disadvantages

[8] Wokwi, "Iot project simulation platform," May 2024. [Online]. Available: https://wokwi.com/

[9] Colblindor, "Color blindness simulator," 2021. [Online]. Available: https://www.color-blindness.com/coblis-color-blindness-simulator/

[10] I. Spirent Communications, "Measuring jitter accurately," May 2007. [Online]. Available: https://piazza.com/class_profile/get_resource/hnd936blk4c26r/hnwio4j6bb5nm#:~:text=If%20the%20received%20packet%20is,delay%20of%20the%20previous%20packet.

Appendix

Fig. A.1 shows the FMEA table.

FAILURE MODE AND EFFECTS ANALYSIS

Item:	HDMI (TX)	Responsibility:	Gurjan, Dhyey, Amin, Alex	FMEA number:	N/A
Model:	PYNQ-Z1	Prepared by:	Amin	Page :	1 of 1
Core Team:	Usual Suspects			FMEA Date (Orig):	12/6/2024 Rev: 1

Process Function	Potential Failure Mode	Potential Effect(s) of Failure	Sev	Class	Potential Cause(s)/ Mechanism(s) of Failure	Occur	Current Process Controls	Detec	RPN	Recommended Action(s)	Responsibility and Target Completion Date	Action Results				
												Actions Taken	Sev	Occ	Det	RPN
Frame Generating Subsystem - Pixel Map submodule	Resolution/Format Mismatch	Incorrect/distorted output	7	N/A	Misconfiguration, firmware bug	5	Operator Instructions	3	105	Change the design of Pixel Map submodule	Amin	14/06/2024	7	2	3	42
HDMI Transmitter Subsystem	Handshake Failure	No signal on display	8	N/A	Timing issues, errors with flags	3	Not implemented yet - Verification of design WaveTrace, Icarus and Verilator	9	216	Set up and run testbenches	Gurjan	12/6/2024	8	1	1	8
Frame Generating Subsystem	FIFO buffer overflow	Video stutter, lag	7	N/A	Insufficient buffer size, high data rate	3	None	10	210	Generate bitstream and test on PYNQ	Alex	12/6/2024	7	1	1	7
Frame Generating Subsystem	Colour Space Conversion Error	Incorrect colours on display	6	N/A	Algorithm bug	3	Not implemented yet - Verification of design WaveTrace, Icarus and Verilator; Test on PYNQ	9	162	Set up and run testbenches	Gurjan	13/06/2024	6	1	1	6
Frame Generating Subsystem	Scaler Malfunction	Distorted or incorrect resolution	8	N/A	Hardware failure, misconfiguration	4	Not implemented yet - Verification of design WaveTrace, Icarus and Verilator; Test on PYNQ	9	288	Set up and run testbenches	Dhyey	14/06/2024	8	1	1	8

Figure A.1: FMEA Table