Interactive Fractal Viewer
CSEE W4840 Final Report

Nathan Hwang - nyh2105@columbia.edu
Richard Nwaobasi - rcn2105@columbia.edu
Luis Peña - lep2141@columbia.edu
Stephen Pratt - sdp2128@columbia.edu

May 9, 2012

Abstract

Fractals are often appreciated for their rich and elegant internal complexity. It is this complexity that is responsible for the beautiful aesthetic of these famed mathematical images as well as the amount of computational power required to generate them. Using fixed point calculations within parallelized sequential logic blocks, we aim to develop an hardware-accelerated fractal generator, capable of computing and displaying quadratic Julia sets in significantly less time than a software-based solution.

1 Background

1.1 An Introduction to Julia sets

The Julia set J of a complex rational function $f : \mathbb{C} \to \mathbb{C}$ is can be expressed as:

$$\forall z : \lim_{n \to \infty} |f^n(z)| = \infty$$

That is, given a function that describes a mapping from the complex numbers to the complex numbers, that function's Julia set is the set of numbers for which repeated application of that function results in convergence towards infinity. Julia sets have many remarkable properties. Iterations of a function is chaotic on its Julia set, meaning that it represents a dynamical system in which tiny perturbations in initial conditions can result in disproportionately large changes in the evolution of the system. As chaotic systems, they can be used to generate pseudo-random numbers. When plotted, Julia sets have the capacity to produce stunning images.

Quadratic polynomial Julia sets are Julia sets of functions that take the form

$$f_c(z) = z^2 + c$$

So any given quadratic polynomial Julia set is uniquely described by some point on the complex plane c. These sets produce fractals, and so they often exhibit self-similarity. Quadratic polynomial Julia sets have the nice property that the magnitude of $f_c(z)$ at any point in a recurrence whose initial value was **not** in the Julia set will always be bounded above by 2. Thus, any point z that causes $|f_c(z_i)| > 2$ for any z_i in the recurrence is necessarily **not** a member of $J(f_c)$

Generating Quadratic Polynomial Julia Sets

Because points not belonging to the Julia set of some quadratic polynomial f_c (also called points in the Fatou set of f_c) are guaranteed to be bounded in magnitude by 2 during repeated application of f_c, one may generate the Julia set for a given window in f_c by sampling the members of that window, repeatedly applying f_c, and testing whether or not the magnitude ever breaks away from 2, giving up after some fixed number of iterations. This is the basic design of the **Interactive Fractal Viewer**.

When plotting the resulting Julia set, one may choose to colorize the image based on how long it took for the iteration beginning at each point in the complex plane to become unbounded. We shall henceforth call this value the breakaway iteration, or k. Thus, the problem of plotting a Julia set on a screen reduces to computing the viewing window as a section of the complex plane, and finding k for each sample point in that section.

Generating Julia sets in this fashion can be a very lengthy process, especially when the points must be done in series. However, since each recurrence to be tested is completely independent of each other, the problem is extraordinarily parallelizeable. Furthermore, since the function being applied is relatively simple, it may be very easily implemented in hardware. Giving the **Interactive Fractal Viewer** its motivation.

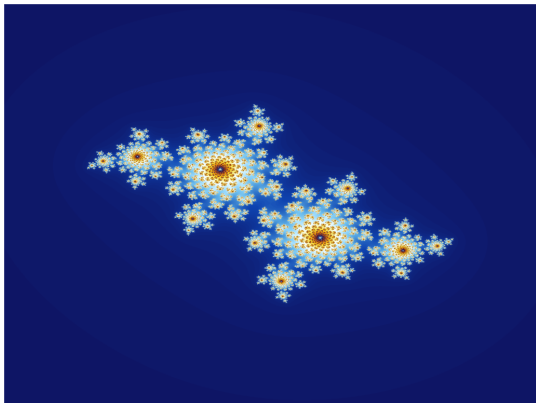

Figure 1: The quadratic polynomial Julia set defined by $c = (\phi - 2) + (\phi - 1)i$ (from Wikipedia)

2 Design

2.1 System Description and Development Environments

System Description

The Interactive Fractal Generator is implemented on an **Altera DE2 Cyclone FPGA** by *Terasic Technologies*. Terasic advertises the following specification about the device:

- Altera Cyclone II 2C35 FPGA with 35000 LEs
- Altera Serial Configuration devices (EPCS16) for Cyclone II 2C35
- USB Blaster built in on board for programming and user API controlling
- JTAG Mode and AS Mode are supported
- 8Mbyte (1M x 4 x 16) SDRAM
- 512K byte(256K X16) SRAM
- 4Mbyte Flash Memory (upgradeable to 4Mbyte)
- SD Card Socket
- 4 Push-button switches
- 18 DPDT switches
- 9 Green User LEDs
- 18 Red User LEDs
- 16 x 2 LCD Module
- 50MHz Oscillator and 27MHz Oscillator for external clock sources
- 24-bit CD-Quality Audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (10-bit high-speed triple DACs) with VGA out connector
- TV Decoder (NTSC/PAL) and TV in connector
- 10/100 Ethernet Controller with socket.
- USB Host/Slave Controller with USB type A and type B connectors.
- RS-232 Transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- IrDA transceiver
- Two 40-pin Expansion Headers with diode protection
- DE2 Lab CD-ROM which contains many examples with source code to exercise the boards, including: SDRAM and Flash Controller, CD-Quality Music Player, VGA and TV Labs, SD Card reader, RS-232/PS-2 Communication Labs, NIOSII, and Control Panel API

Development Environments

- VHDL Development was done in the Altera Quartus II IDE v. 7.2
- Software Development was done in a combination of Emacs v. 23.1 and the NIOS II IDE

2.2 High-level Overview

The following is a description of how data travels through the block structure specified in Figure **??** when generating a single fractal.

- Data flow originates from the **NIOS II Processor**, which initializes the system by computing a set of parameters that can be used to describe the target window and fractal. The **NIOS** writes these parameters across the **Avalon Bus** onto an on-board **RAM** and sends instructions to generate a new image to a special instruction register.
- When the instruction register receives the generate signal, it tells a component charmingly referred to as the `Rammer` to write the parameters information contained in the **RAM** into registers. It then asserts a *generate* signal that will be read by the `Window Generator`
- The `Window Generator` takes these parameters as input and builds a set of 4-tuples (x, y, a, b) where each tuple is a mapping from a **VGA** coordinate (x, y) to a value in the complex plane of the form $a + bi$. The (a, b) values eventually be used to compute a breakaway count k for each co-ordinate on the screen. The computation will be performed by a specialized component called an `Iterative Function Module`, or `IFM`.
- Before the next (a, b, x, y) tuple is delivered to one of the 4 available `IFM`s, it must be requested by a component known as the `IFM Controller`. The `IFM Controller` is responsible for distributing work among several `IFM`s working in parallel.
- When an `IFM` enters a ready state, it is given the next (a, b, x, y) tuple and will begin performing the function iterations using the constant prescribed by the set to be generated. The `IFM` transitions into a done state after a fixed number of iterations, or when the squared magnitude of its current iteration exceeds 4.
- The `IFM Controller` takes the data given by the done-state `IFM`s and writes it to a queue that will deliver it to the `Coordinate-Breakaway Lookup Table`, implemented using the on-board **SRAM**. The `IFM` data takes the form of the (x, y, k) triples we set out to create. The k value of each triple is stored in the `Coordinate-Breakaway Lookup Table` at an address determined by the (x, y) values. In this way, we map **VGA** coordinates to their associated breakaway iterations.
- The `VGA Module` fetches results from the `Coordinate-Breakaway Lookup Table` and colorizes them using a separate **ROM**-based `Colorization Lookup Table`. The `Colorization Lookup Table` takes a breakaway count, k, and maps it to an (r, g, b) bit-vector for use by the **VGA**.

2.3 Module Implementation

User Interface Module

As an interactive device, our fractal generator has the capacity to accept user parameters such as window size or Julia set constants during operation. This communication with the user is facilitated by the NIOS II Processor taking PS/2 keyboard input. We refer to these entities and all their supporting devices as the `User Interface Module`. This module is responsible for handling communication with input peripherals and translating user input into information that can easily be used by the hardware-based fractal generator. Once this information has been translated into a set of values that can be used by the remainder of the system generator, they are written into an on-board RAM. When the `User Interface Module` is ready for the hardware to take some action, it sends a signal to a component on the board, which forwards the signal appropriately. All communication with the board is performed over the Avalon Interconnect Fabric.

The NIOS II Processor uses the SDRAM as its memory store.

Parameters of primary concern are those of viewing window and Julia set constant. The window is set using the following values (which will be elaborated on in the `Window Generator` section):

- a_{min} 36 bits
- a_{diff} 36 bits
- a_{leap} 10 bits
- b_{min} 36 bits
- b_{diff} 36 bits
- b_{leap} 10 bits

Meanwhile, the Julia set constant is set using the following values

- c_{real} 36 bits
- c_{img} 36 bits

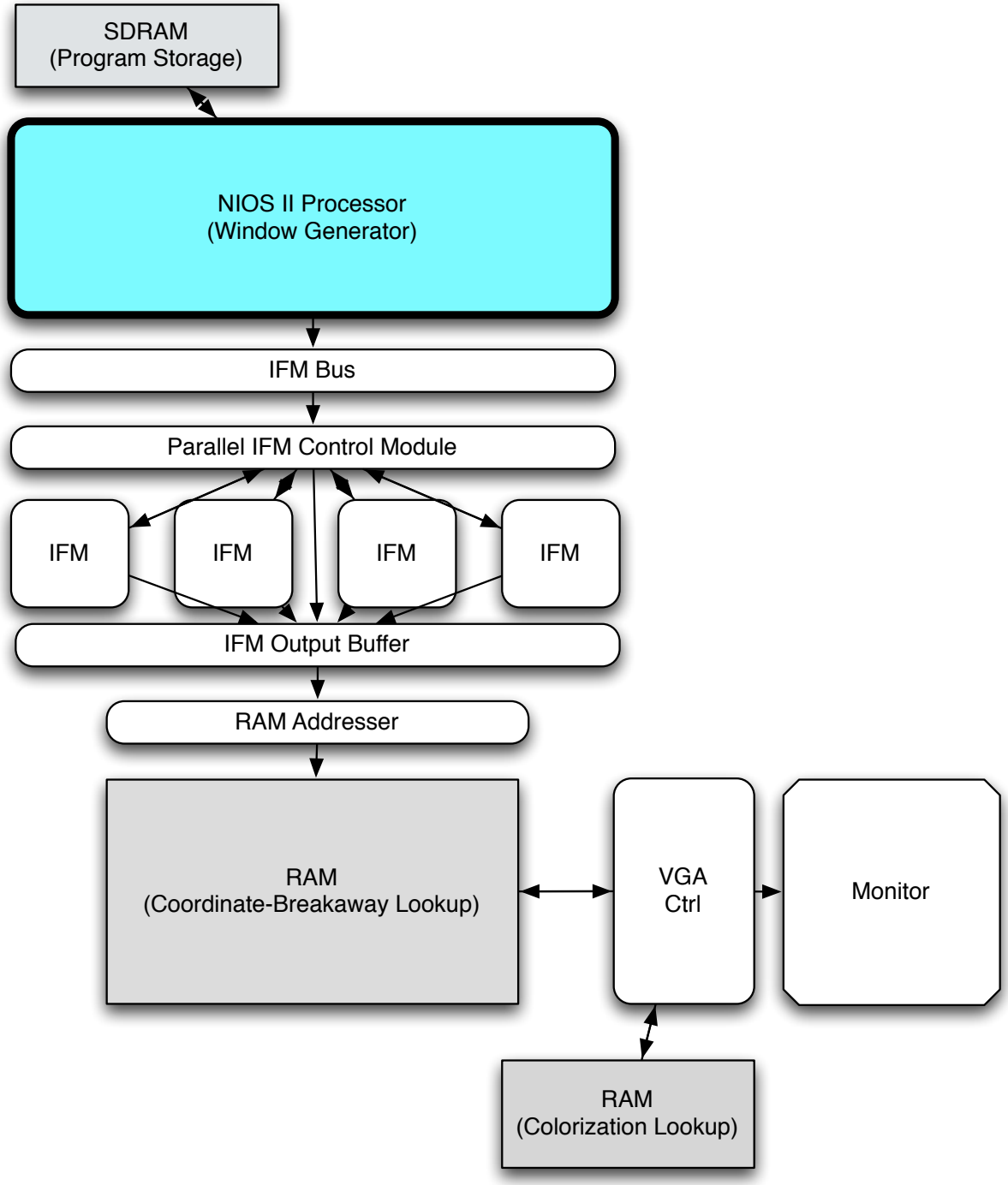

Figure 2: High-level Block Diagram

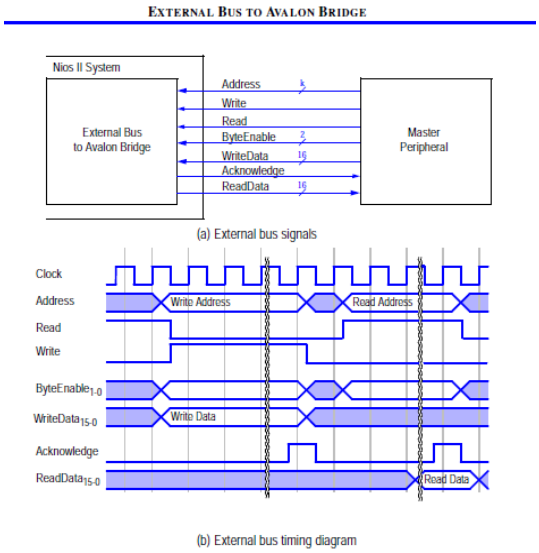

Figure 3: Block and Timing Diagram of the Avalon interconnect fabric. Provided by Altera Corporation.

Parameter RAM

In order to allow for the buffering and mutation of individual parameters, the `UI Module` writes each individual parameter into a RAM. Since the Avalon interconnect fabric only allows for 32 bits to be transferred at once, but most parametrs are 36 bits wide, the RAM is 18 bits in width and parameters are sent in with the 18 least significant bits of each 32 bit write. For simplicity, these rows are still addressed by the NIOS as if they were 32 bits wide.

Each of the 14 rows in this RAM has a different purpose.

Row	Data Stored	Portion	Offset
0	a_{min}	18 MSB	0x0
1	a_{min}	18 LSB	0x4
2	b_{min}	18 MSB	0x8
3	b_{min}	18 LSB	0xC
4	a_{diff}	18 MSB	0x10
5	a_{diff}	18 LSB	0x14
6	b_{diff}	18 MSB	0x18
7	b_{diff}	18 LSB	0x1C
8	a_{leap}	-	0x20
9	b_{leap}	-	0x24
10	c_{real}	18 MSB	0x28
11	c_{real}	18 LSB	0x2C
12	c_{img}	18 MSB	0x30
13	c_{img}	18 LSB	0x34

When the `UI Module` sends a redraw signal to the board, a device called the `Rammer` moves these values into registers that will be read by the `Window Generator` and `IFM`s when drawing the image.

Window Generator

The `Window Generator` serves to kick off the calculation cascade, computing the position in the complex plane represented by each pixel in the given input window, thereby producing (x, y, a, b) tuples.

The generator uses a specialized procedure that requires only addition and comparison operations to map out a whole window. Say we have a window that stretches from v_{min} to v_{max} over N pixels. The procedure works by iterating from 0 to N-1 and producing a sum at each step of the way that corresponds to a the value of v at that point. The procedure requires a few values as input:

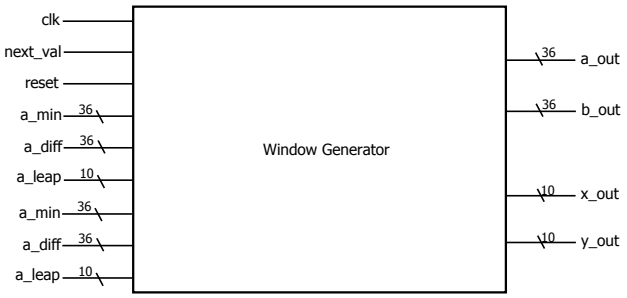

Figure 4: High-level Block Diagram of the Window Generator

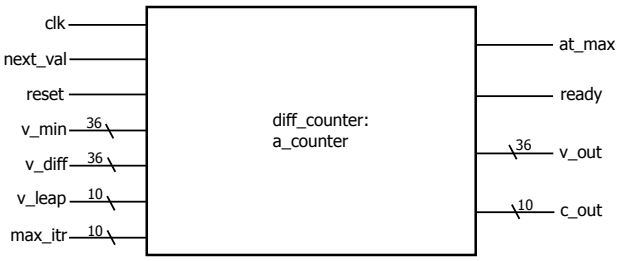

Figure 5: Block diagram of a `Differential Counter` used in window generation

Parameter	Width	Purpose
a_{min}	36 bits	Describes the minimum value of a in the window
a_{diff}	36 bits	Describes the standard differential between consecutive a values in the window $(a_{max} - a_{min})/WIDTH_{SCREEN}$ this is computed by the NIOS processor.
a_{leap}	10 bits	Periodically, we will need to add 1 to our sum to compensate for precision loss. This value corresponds to the length of the intervals between these "leap cycles" $WIDTH_{SCREEN}/((a_{max} - a_{min}) \bmod WIDTH_{SCREEN})$
b_{min}	36 bits	Describes the minimum value of a in the window.
b_{diff}	36 bits	Describes the standard differential between consecutive a values in the window $(b_{max} - b_{min})/HEIGHT_{SCREEN}$ this is computed by the NIOS processor.
b_{leap}	10 bits	Periodically, we will need to add 1 to our sum to compensate for precision loss. This value corresponds to the length of the intervals between these "leap cycles" $HEIGHT_{SCREEN}/((b_{max} - b_{min}) \bmod HEIGHT_{SCREEN})$

The `Window Generator` is therefore comprised of two `Differential Counters` that are responsible for performing the iterations. One computes values for b and the other a.

When the `Differential Counter` receives a reset signal, it initializes its data according to the signals coming in. Then, each time it recieves a next-value signal, it increments the output value accordingly. If the counter reaches its maximum, it asserts a flag.

In the `Window Generator`, the `Differential Counters` are hooked up in such a way that the points are cycled through from left to right down the screen.

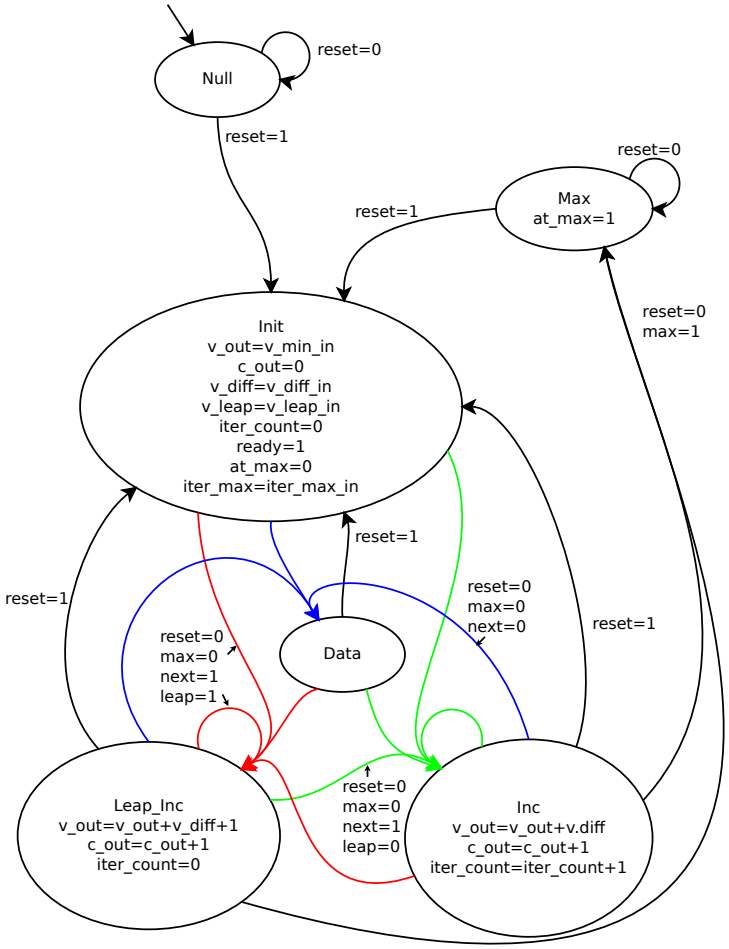

Figure 6: State Diagram for the `Differential Counter`, Moore machine: signals `max=c_cuis=max_itr`, `leap=iter_count=v_leap`. Omit unused signals (X) for compactness. Colorcoded signal bundles.

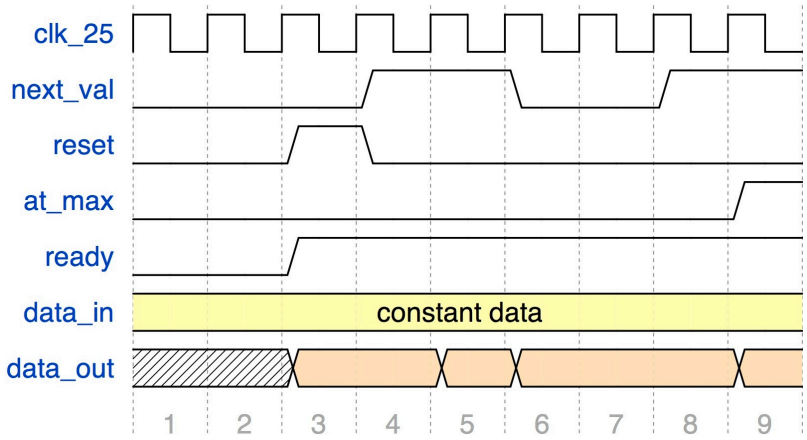

Figure 7: Timing diagram of the `Differential Counter`

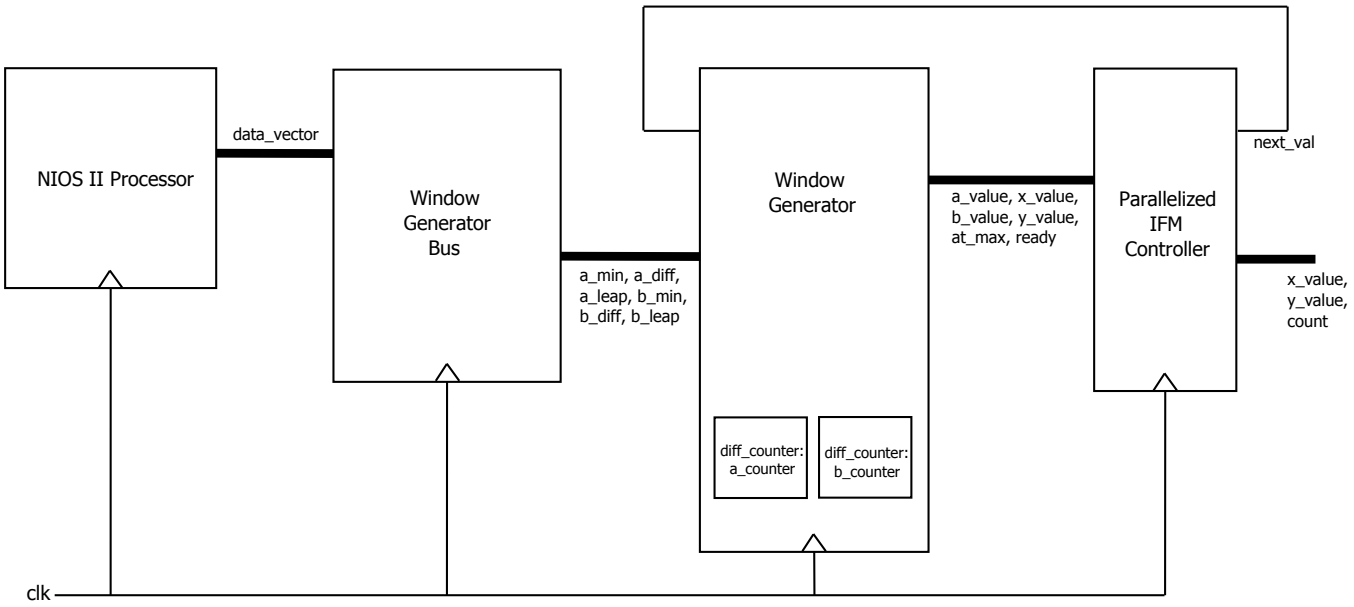

Figure 8: Block diagram illustrating (x, y, a, b) tuple dataflow.

The `Window Generator` takes reset signals from the bus connected to the NIOS processor. When the `Window Generator` is ready for computation (the same cycle that it is reset by the bus), it asserts a data flag. When the `IFM` reads an (x, y, a, b) tuple, it asserts a next value signal indicating that it will need new data in the following cycle. Once the `Window Generator` runs out of values to give, it asserts an at max flag.

IFM Controller

Rendering Julia set fractals requires many iterations of relatively simple computations in the complex plane. This sequence of computations is independent for each point in the image, which is why the calculation of fractal sets lends itself to parallel computation. However, the very nature of the iterated fractal calculation means that the amount of time spent performing computations on each individual point can vary drastically, introducing synchronization issues. It is the responsibility of the `IFM Controller` to resolve these issues.

The `IFM Controller` constantly transmits the (x, y, a, b) tuple currently being expressed by the window generator to each of the `IFM`s. When an `IFM` indicates that it is in the ready state, the controller asserts a signal instructing the `IFM` to accept the new data and begin computation (assuming that the window generator is asserting the valid data flag). Simultaneously, the controller signals to the `Window Generator` that it needs the next data tuple in the window. If more than one `IFM` is in the ready state at once, the controller only sends the read and compute signal to one, saving the upcoming data tuples for the rest.

Iterative Function Module (IFM)

A quadratic polynomial Julia set is generated by applying the function

$$f_c(z) = z^2 + c \tag{1}$$

repeatedly, where $z, c \in \mathbb{C}$. For any given pair (z, c), this recurrence will result in one of two outcomes:

- The magnitude of the complex values generated by the recurrence may stay bounded by 2
- The magnitude may become unbounded and escape toward infinity

A point z on the complex plane is in the Julia set uniquely defined by the complex number c if and only if the recurrence remains bounded for (z, c). To determine whether or not a point remains bounded

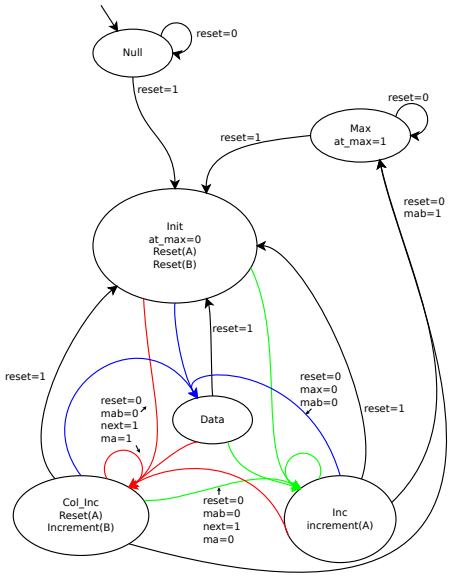

Figure 9: State Diagram for the `Window Generator` code, Moore machine: signals `max=c_cuis=max_itr`, `leap=iter_count=v_leap`. Omit unused signals (X) for compactness.

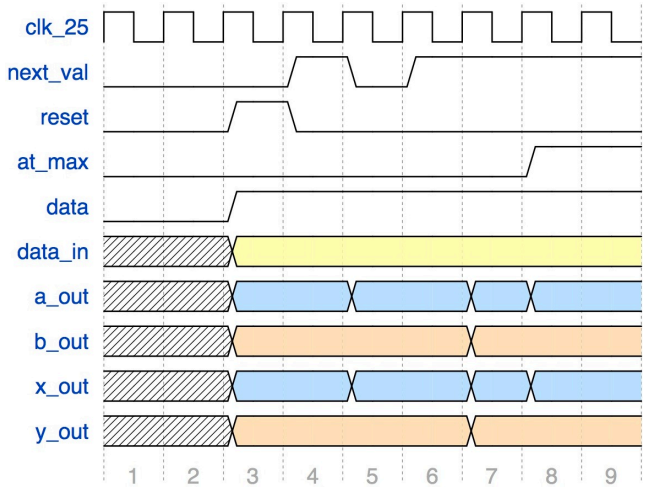

Figure 10: Timing diagram of interface between the `Window Generator` and the IFMs.

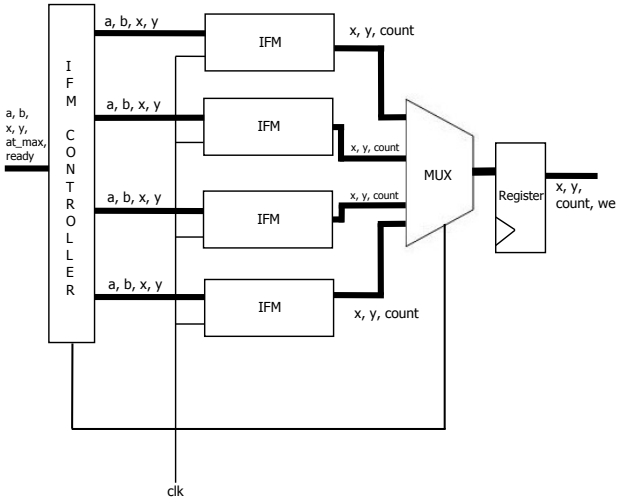

Figure 11: Block diagram for the IFM wrappers

for a given c, we compute a fixed number of iterations on the recurrence (in our case 127) and report the iteration in which the value generated has a squared magnitude of greater than 4. Those points that do not become unbounded in this many iterations are considered to be part of the set.

Because the factors of the multiplication are complex numbers, computing their product involves 3 real-number multiplications. For $z = a + bi$ we compute

$$\begin{aligned} P_A &= a^2 \\ P_B &= b^2 \\ P_C &= ab \end{aligned}$$

With these values we can compute:

$$\begin{aligned} a_{next} &= P_A - P_B + c_{real} \\ b_{next} &= 2P_C - c_{img} \\ |z|^2 &= P_A + P_B \end{aligned}$$

The squaring operations for P_A and P_B is perfomed by a specialized logical circuit provided as an *Altera Megafunction*. The multiplication P_C is perfomed by embedded multipliers on the DE2. These are expansive circuits, but the FPGA is still to accomodate up to 4 `IFM`s.

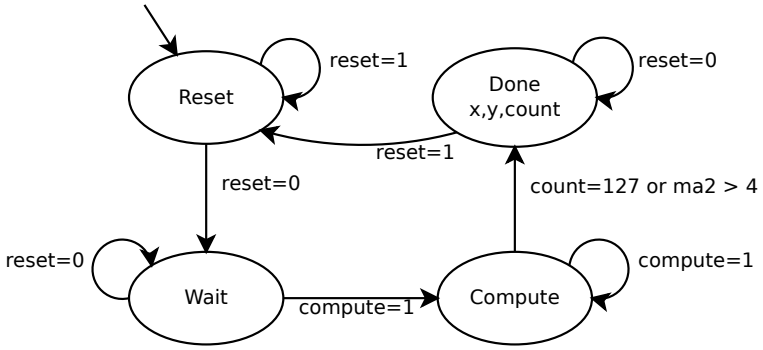

Figure 12: State diagram for a single `IFM`. Moore machine: using abstract transition descriptions. Omits unused signals for compactness.

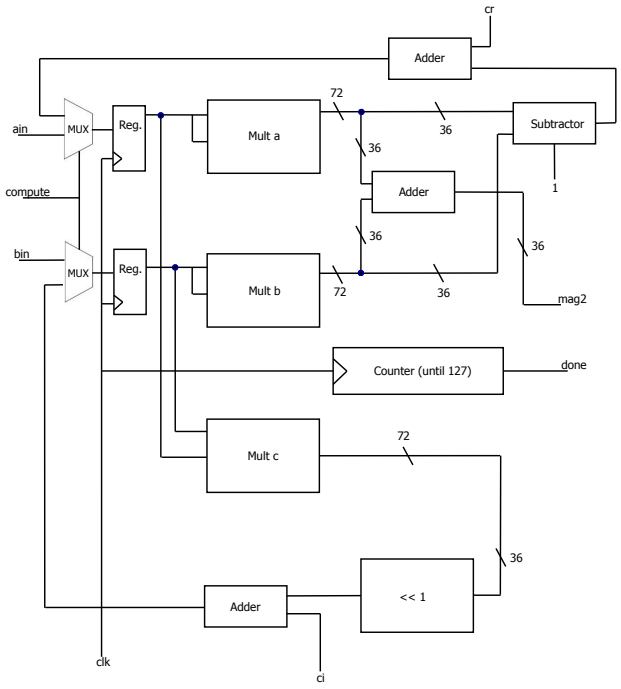

Figure 13: Arithmetic Logic Circuit within each `IFM`

Real-valued numbers are represented as two's-complement fixed-point binary values in our circuit. A complex number is comprised of two such data vectors. We restrict ourselves to 36 bits, as the onboard multipliers are sized as such.

In order to accomodate the largest-magnitude value we expect to come across during any iteration, we require 6 bits to the left of the radix. Thus, our fixed-point values have 30 bits to the right of the radix. This gives us a machine precision of approximately $9.31 \times 10^{-}10$.

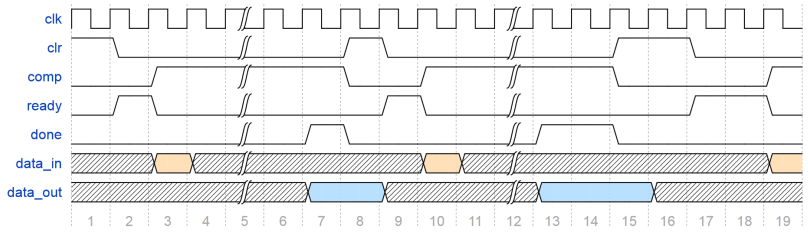

Figure 14: Timing diagram for a single `IFM`.

To more easily facilitate communication with the `IFM Controller`, each `IFM` is contained within a wrapper module. Thus, the `IFM Controller` need only alter the state of the wrapper module, and the wrapper module will transmit signals to the `IFM`s indicating the desired behavior.

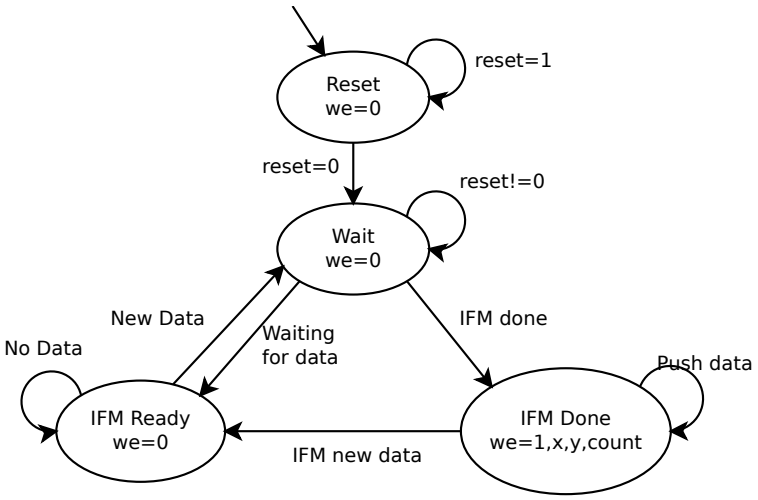

Figure 15: State diagram for a single `IFM` wrapper module, Moore machine: using abstract transition descriptions.

If a wrapper module is in the done state, the controller indicates that its (x, y, c) triple should be read into the output register. If multiple `IFM` wrappers are in the done state simultaneously, the controller chooses one at a time to be read in. These triples are then augmented with an asserted write enable flag to indicate that they represent valid data, and should be written to the `Coordinate-Breakaway lookup table`.

Coordinate-Breakaway Lookup Table

After the count associated with each pixel is calculated, it must be stored in a framebuffer that interfaces both with the `IFM Controller` as well as the `VGA Module`.

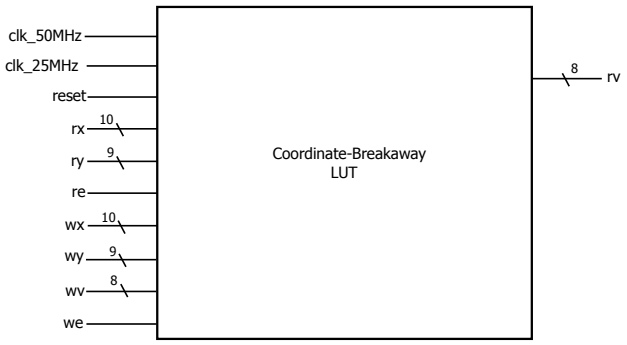

Figure 16: Block Diagram of the `Coordinate-Breakaway LUT`: signals on the right side are inputs, signals on the left are outputs.

For this, we use the SRAM chip that is built into the DE2 board, for its relatively expansive memory size (versus on-chip memory), fast speed, and ease of use (versus the SDRAM chip). The SRAM chip has a 512kibibytes capacity that can be accessed and written to in half a 50MHz clock cycle, making it ideal for our purposes.

Since we display a 640×480 image in the VGA module and keep 8 bits of iteration information for each pixel, we need a grand total of 300kibibytes to store the information, fitting well within the confines of the given 512kibibyte SRAM chip.

We use a straightforward addressing scheme to store the count information, using the y position as the top 9 bits of the address, and the x position as the bottom 10 bits of the address. This way, finding the address from a given pixel position is very fast.

A small wrinkle is the fact the SRAM is in fact a $256K \times 16$ bit memory, reading and writing in 16 bit chunks. This merely means that the very bottom bit of the x position does not go to the address, but is routed to the bitmask signal indicating whether the byte sought is in the upper or lower half. Of the 16-bit word that is addressed by the remaining 18 bits.

Reading/Writing Since the SRAM has only one IO port, reads and writes must be time multiplexed. The VGA module will be consistently requesting data from the SRAM at 25MHz. However, while the fractal is being generated, the IFMs will be providing information that must be written to the SRAM at the same frequency. This means that we must interleave reads and writes to the SRAM.

We can use the structure of the reads from the VGA to our advantage to make room for the necessary writes. Reads always follow a pattern, where if we read the lower half of a 16bit word, then we will read the higher half in the next 25MHz clock cycle. Hence, when we require the lower half of a word, we can fetch the entire word in one read, save the higher half in a register, and return it when it's required in the next clock cycle. In this way, we reduce the frequency of VGA reads from the SRAM to every other cycle on a 25MHz clock.

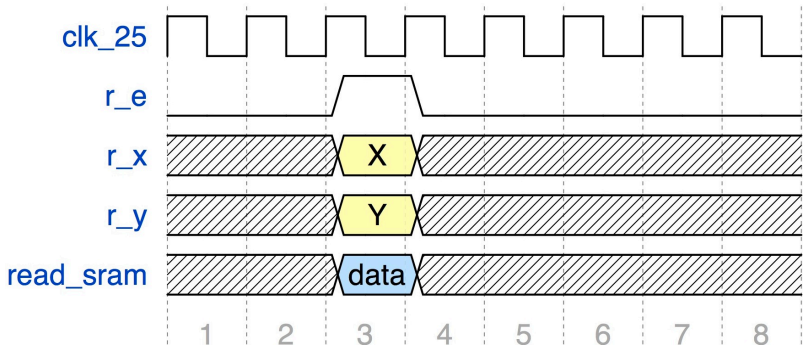

Figure 17: Timing diagram of the interface between the Coordinate-Breakaway LUT and VGA module

Even with every other 25Mhz cycle being dedicated to writing the data being sent out by the `IFM`s, the SRAM might still miss a coordinate if the `IFM`s are generating their maximum possible throughput of 25MB/s. To account for this, we put the junction serving writedata to the SRAM on a 50MHz clock. This junction consists of a shift register that constantly reads from the `IFM` output, but only shifts when the SRAM's read enable signal is not being asserted.

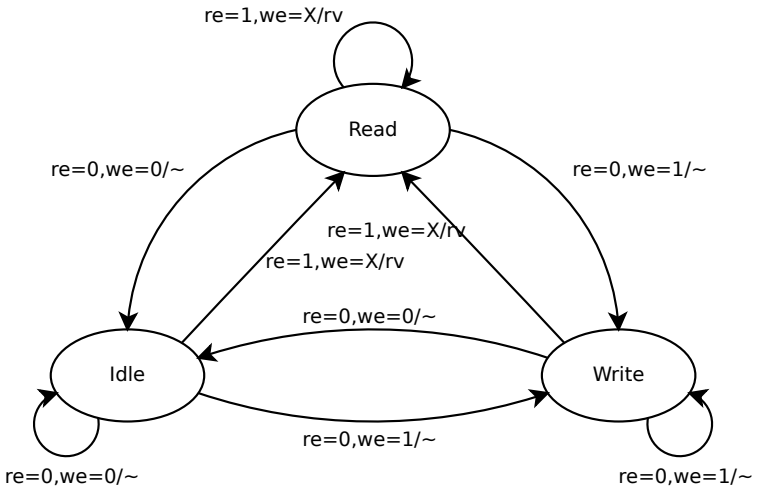

Figure 18: Simplified State Diagram of the `Coordinate-Breakaway LUT`: Mealy machine, only includes re and we as inputs and rv as an output, with $\sim$ denoting a lack of outputs

This means that a given tuple (x_j, y_j, c_j, we_j) being output by the IFM controller will be overwritten by the following (x_k, y_k, c_k, we_k) output during every VGA read cycle. However, since the shift register is reading from a 25MHz process at 50MHz, it is guaranteed that every write tuple will be read into the shift register twice. Because the IFM and VGA controllers are synchronized, the VGA read always occurs when $j \neq k$. But because every tuple is read into the shift register twice, the value that was overwritten, (x_j, y_j, c_j, we_j), must also exist in the next cell over, guaranteeing reliable data transmission.

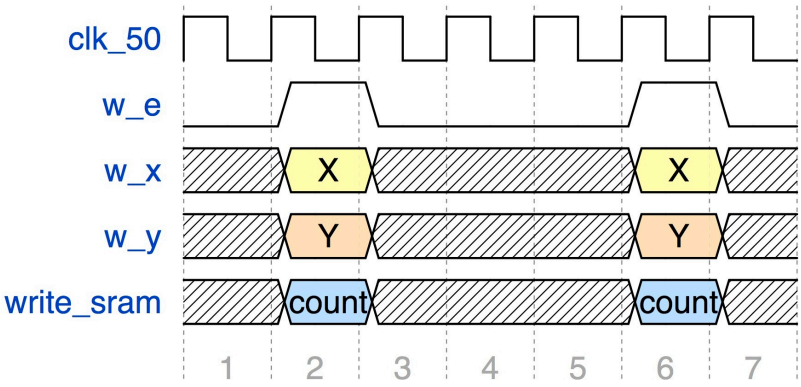

Figure 19: Timing diagram of the interface between the `IFM`s and the `Coordinate-Breakaway LUT`.

Dangerous though it may be, the effect of having a faster clock for the SRAM write junction is well contained. No processes depend on the state of the write junction, and the junction is free to produce redundant write data without consequence. The write junction merely serves as a conduit through which writedata is transmitted.

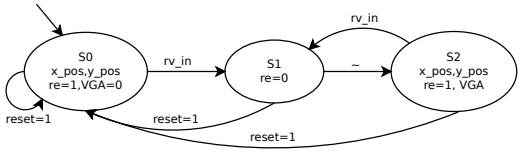

Figure 21: State diagram for the `VGA module`, Mealy machine: $\sim$ stands in for no input/output, and VGA stands in for all the `VGA_` signals (`VGA_CLK`, `VGA_HS`, `VGA_VS`, `VGA_BLANK`, `VGA_SYNC`, `VGA_R`, `VGA_G`, `VGA_B`). Omits unused signals for compactness.

VGA Module

In order to display the generated Julia set, we connect a VGA controller to the `Coordinate-Breakaway lookup table`. As the controller cycles through output coordinates within the display area, it modifies the read address signal for the lookup table. The data signal coming from the RAM is thus the breakaway value associated with that coordinate.

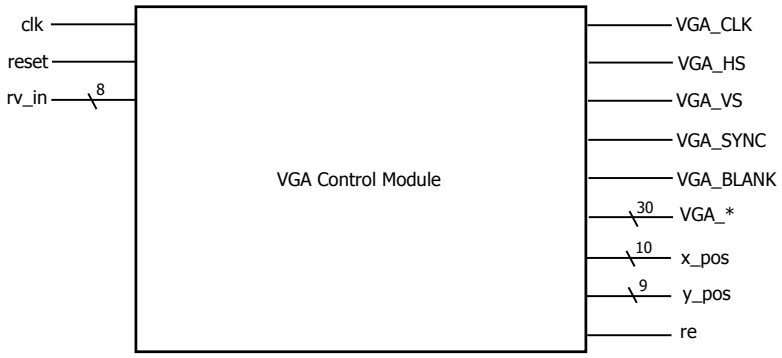

Figure 20: Block diagram of the VGA module

This breakaway value is passed through a decoder known as the `Colorization lookup table` and the resulting (R, G, B) signal tuple is sent to the VGA port.

Colorization lookup table

The `Colorization lookup table` is implemented using a ROM on the FPGA. The ROM maps each possible breakaway k value to a bit-vector corresponding to the (R, G, B) signal tuple that should be expressed for that value. The system includes a special "cycle colors" mode, wherein the k value being sent in to the `Colorization lookup table` is modified by a linearly increasing amount. This causes the colors to shift on screen, creating a remarkable aesthetic.

To assist with programming the ROM, the development team created a small Python application that allows the user to graphically modify (R, G, B) components for different k values, and interpolates the results. The application can then display a preview of what a Julia set will look like with these settings, and produces VHDL for programming the ROM accordingly.

3 System Validation and Performance

3.1 Verification and Validation

During development of the Interactive Fractal Viewer, it was desireable to validate the functionality of individual components. We used VHDL Test Benches in combination with the Quartus RTL simulation tool to verify proper timing and behavior during the early stages of development.

The development team wrote Test Benches for each of the major hardware components. These proved invaluable in finding problems during the initial implementation stages, as well as diagnosing them later on.

Additionally, in order to strictly verify the correctness of the Julia sets that the system produced, the team wrote a specialized Test Bench capable of dumping (a, b, k) triples to a file. A Python script was written to parse and validate this file. The script built a complex number for each (a, b) double and computed its own breakaway iteration k' for that value using floating point precision. If the $|k-k'| < \epsilon$, it marks the value as a success. Otherwise, it is marked as a failure. The tolerance parameter ϵ is necessary because Julia set iterations represent chaotic systems. Therefore, the minor perturbations caused by the difference in hardware vs. software computation can cause major changes.

For the Julia set described by a chosen c, and $\epsilon = 5$, the validation script showed a 98% success rate on the 307200 points tested.

3.2 Performance Analysis

Upper-Bound on Image Generation Time

Say the system wishes to compute an arbitrary window of an arbitrary Julia set. The only variable in generation time for a given image is the amount of time it takes for the `IFM`s to completely write all k values to the `Coordinate-Breakaway lookup table`. Thus, to find an upper bound on image generation time, we may assume that each `IFM` spends the maximum possible amount of time computing the k value for each coordinate. The number of iterations in each `IFM` is capped at 127. Each `IFV` can perform one function evaluation per clock cycle, and requires 3 cycles of setup and tear-down time for state changes.

Additionally the `IFM Controller` takes a clock cycle between each `IFM` value assignment to perform a check for a ready `IFM` and grab a new value from the `Window Generator`.

Thus, the system sets up a pipeline extending from the `IFV Controller` to the `Coordinate-Breakaway lookup table`. When using 4 `IFM`s for computation, it may achieve a throughput of $\frac{4 \text{ values}}{133 \text{ iterations}}$

Though the system has to worry about scheduling conflicts from the controller on the first set of values to go to the `IFM`s (since only one `IFM` may be assigned or report a value at a once), this issue vanishes due to pipelining on subsequent value sets because maximizing the number of iterations required for each value means synchronizing the start and end times of each `IFM`.

It is now possible compute the delay incurred between the `IFM Controller` and the `Coordinate-Breakaway lookup table` when generating a full image on a 25MHz clock in this scenario. $640 \times 480 = 307200$ values must be computed, so

$$\frac{307200 \text{ values}}{1 \text{ image}} \times \frac{133 \text{ iterations}}{4 \text{ values}} \times \frac{1 \text{ cycle}}{1 \text{ iteration}} \times \frac{1 \text{ seconds}}{25000000 \text{ cycles}} = \frac{0.408576 \text{ seconds}}{\text{image}}$$

Which is a good approximation of the upper bound on total time it takes for data to flow through the system.

Experimental Results

Simulations on a variety of constants confirmed the above results, producing delays ranging from $0.125s$ to $0.250s$ on use case images, and a delay of $0.395s$ on a "maximum cost" image.

4 Milestone Report

Milestone	Date	Goal	Accomplished
Milestone 1	Mar 27	Have a static Julia set filled into a buffer.	Develop a `Window Generator` capable of communicating with a parallelized `IFM Controller`
Milestone 2	Apr 10	Display the colorized Julia set through VGA.	*As planned.*
Milestone 3	Apr 24	Implement parameter mutation, with subsequent updates to the displayed Julia set.	Hardware-based parameter mutation and set redraw.
Deadline	May 9	Feature-complete system with accompanying report and presentation.	*As planned.*

5 Contributions and Teamwork

As in all good engineering projects, our development team was heavily collaborative. There are very few files in our source that were produced by one person alone. However, our modular development process allowed different team members to accept primary responsibility for certain files. The following is a list of major contributions of each team member to the project

- **Nathan Hwang**
  1. Project management work
  2. `Coordinate-Breakaway lookup table`
  3. VHDL for color cycling
  4. Instruction registers for communication with NIOS
- **Richard Nwaobasi**
  1. `Colorization lookup table`
  2. `VGA Module`
  3. Communication with PS/2 keyboard
  4. Control loop software (with Stephen Pratt)
- **Luis Peña**
  1. Systems integration
  2. Top level module
  3. `IFM` and `IFM Controller`
  4. Parameterization RAM (with Stephen Pratt)
- **Stephen Pratt**
  1. Document composition and presentaiton management
  2. Integration test bench and validation script
  3. `Window Generator` and window generation algorithm
  4. Parameterization RAM (with Luis Peña)
  5. Control loop software (with Richard Nwaobasi)

6 Challenges and Lessons Learned

This project was one filled with learning opportunities, both in terms of technical knowlege as well as personal growth. Some major design and implementation challenges included:

1. **Design and Implementation of the IFMs** - Working with a system as parallel and dynamic as our `IFM` array presented a very unique set of challenges. Many communication protocols, state machines, and design concepts were discussed when planning the design of this core component of our device. Because most elements of our system needed to interface with the `IFM Controller` in some way, successful integration of the component was in many ways a cornerstone acheivement for our team.

2. **Maintaining Timing Discipline** - Clocks were consistently an adversary for our team, as many parts of our system needed to operate at different rates. The NIOS processor and the buses it writes to necessarily run at 50MHz, while the critical path on our `IFM`s allowed for only 25MHz frequencies. The SDRAM clock needed to lag the NIOS to account for setup, charging, and hold times, and the VGA clock needed to run at just over 25MHz to meet protocol. We did run into trouble on many occasions when trying to manage interplay between these components, but this taught us a lot about the importance of synchronization in Embedded Systems and the nature of Phase-Locked Loops.

Furthermore, each group member ended up with a few personal takeaways:

- **Nathan Hwang**
  1.
- **Richard Nwaobasi**
  1.
- **Luis Peña**
  1.
- **Stephen Pratt**
  1. Consistent progress is of central importance to the success of a project. A team should always focus on moving forward at least a little bit, no matter how busy the week. Even the most marginal rates of progress are far preferable to the overhead that ends up getting poured into teardown/setup time if a team decides to focus its efforts elsewhere for even a short period.
  2. Implementation challenges are not exam questions. They do not need to be solved alone, or even successfully on the first go around. It's far better to experiment and utilize resources that are available than to design a failing solution and pour time into trying to make the implementation work out.

7 Reflections and Prospective

All things considered, our team is extremely satisfied with the results that we've achieved. Development was certainly not without its ups and downs, but we ultimately brought all major project goals to fruition.

Of course, like all good engineers, we can never claim to be completely satisfied with the fruits of our labor. The following are a few features that we would like to implement once we have more time to work with the project:

1. Though window and constant selection are currently software-mutable system parameters, the UI through which these parameters are modified does not allow very intuitive control of their values. A primary reason for this is that our original system architecture was built with the explicit purpose of drawing fractals on the screen, not anything else. In the future, we would like to let users select a custom window by modifying a frame on-screen. Explicit, rather than relative specification of new seed constants for the Julia set would also be desirable.
2. There exist a few timing glitches that were never completely resolved. Ironing out these glitches is a high priority for future work.
3. It is our professional opinion that the color-cycle mode is totally radical. Having this cycling rate vary with audio input would turn our Interactive Fractal Viewer into a spectacular music visualizer.

A Source Code

A.1 VHDL

ifv.vhd

ramcon.vhd

rammer.vhd

window_gen.vhd

diff_counter.vhd

hook.vhd

ifmd.vhd

ifmunitd.vhd

sram.vhd

vga.vhd

vgamod.vhd

test_filename.vhd

Other VHDL Sources / Libraries

A.2 C

ifv.c

ps2_keyboard.h

ps2_keyboard.c

ps2_up_ps2_port.h

ps2_up_ps2_port_regs.h

Other C Sources / Libraries

A.3 Python

julia_gen.py

test_ifm.py

test_ifm_utils.py

test_image.py

color_gen.py

Other Python Sources / Libraries