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 8, 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.

2 Design

2.1 High-level Overview

The following is a description of how data travels through the block structure specified in Figure 1.1 when generating a single fractal.

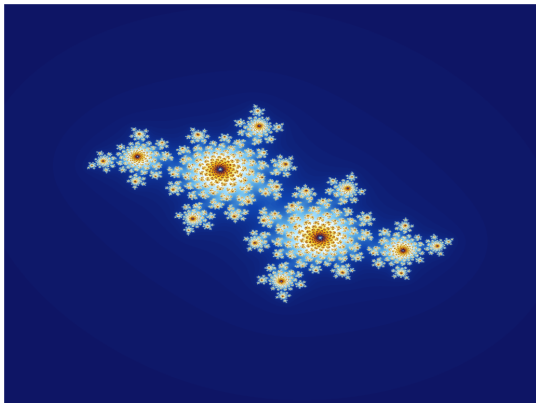

Figure 1: The quadratic polynomial Julia set defined by $c = (\phi - 2) + (\phi - 1)i$ (from Wikipedia)

- 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 an `IFM`, 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**.

2.2 Critical Modules

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 UI module, which is implemented on the NIOS II processor. 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 instructions for the generator, these instructions are written to a specialized bus across the Avalon interconnect fabric.

We have the Nios II configured to use the SDRAM as its memory store, making the SRAM available for other uses.

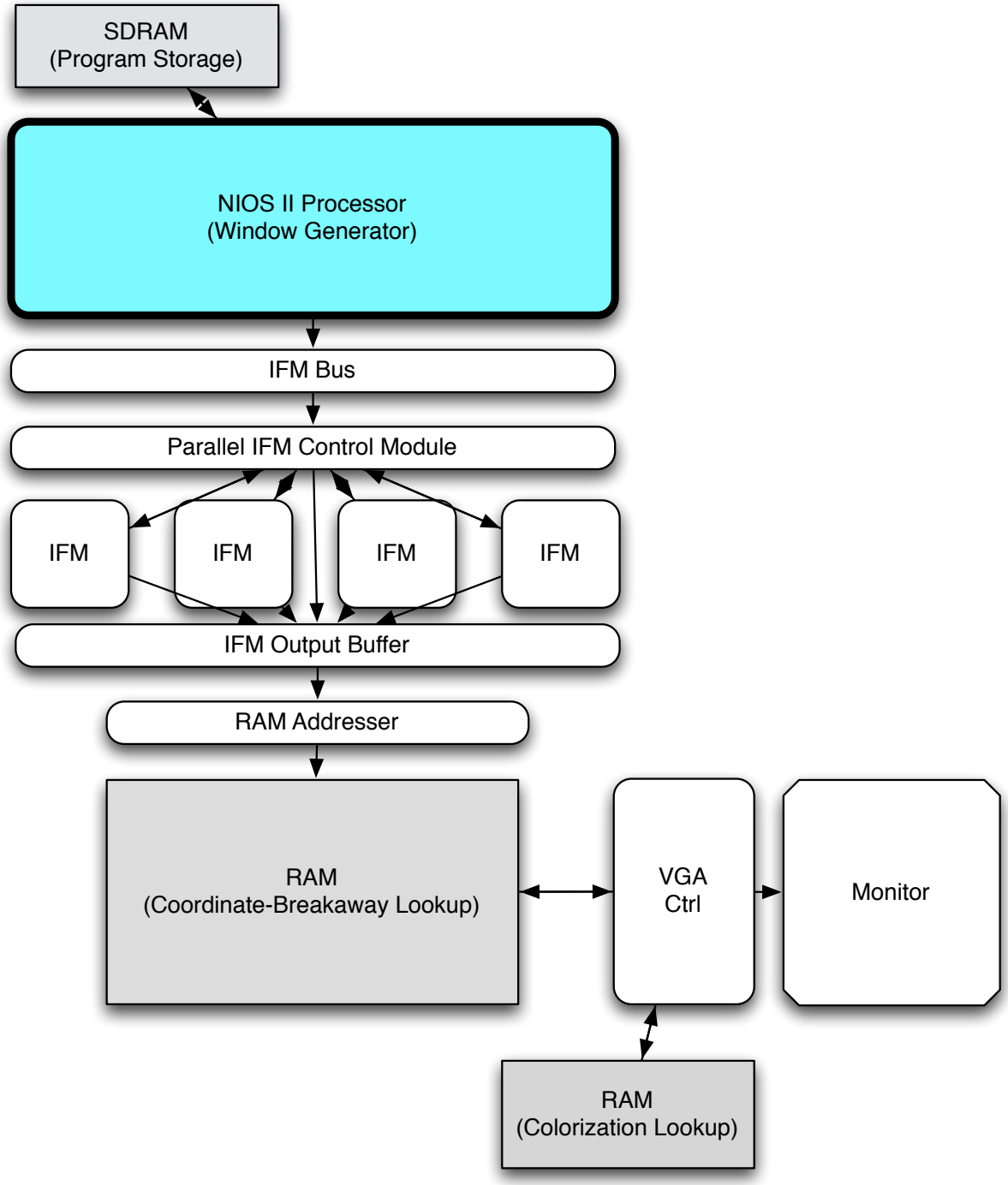

Figure 2: High-level Block Diagram

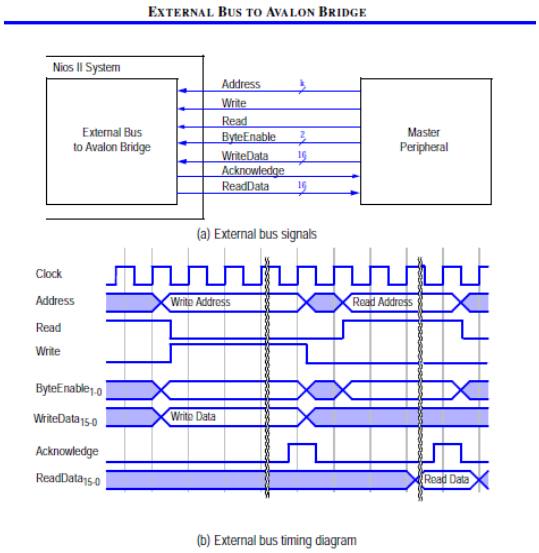

Figure 3: Block and Timing Diagram of the Avalon interconnect fabric. Provided by Altera Corporation.

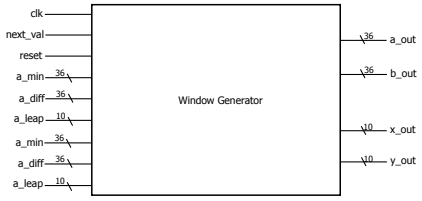

Figure 4: High-level Block Diagram of the Window Generator

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 next 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

New RAM stuff goes here

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:

- 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.

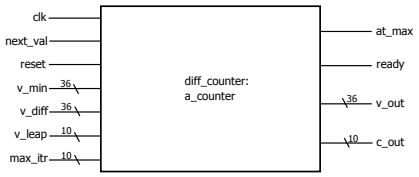

Figure 5: Block diagram of a differential counter used in window generation

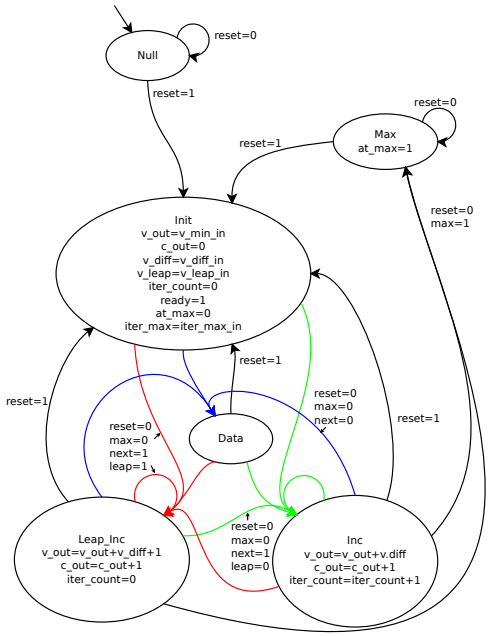

Figure 6: State Diagram for the diff counter, Moore machine: signals `max=c_cuis=max_itr`, `leap=iter_count=v_leap`. Omit unused signals (X) for compactness. Colorcoded signal bundles.

- $a_{leap} - 10$ 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" $WIDTH_{SCREEN}/((a_{max} - a_{min})\%WIDTH_{SCREEN})$
- $b_{min} - 36$ 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" $HEIGHT_{SCREEN}/((b_{max} - b_{min})\%HEIGHT_{SCREEN})$

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.

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.

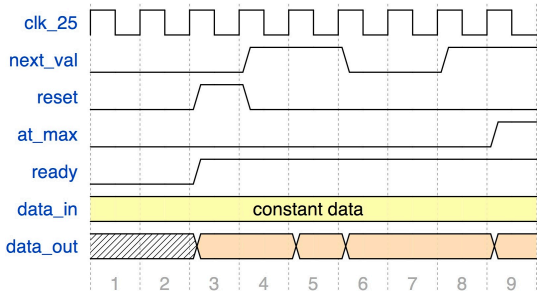

Figure 7: Timing diagram of the differential counter

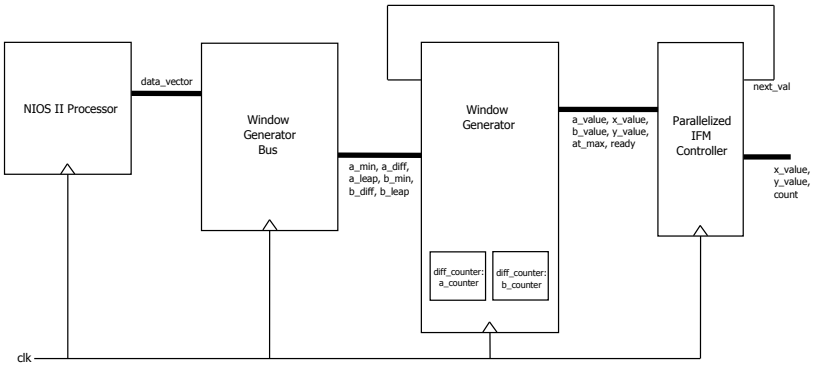

Figure 8: Block diagram illustrating (x, y, a, b) tuple dataflow.

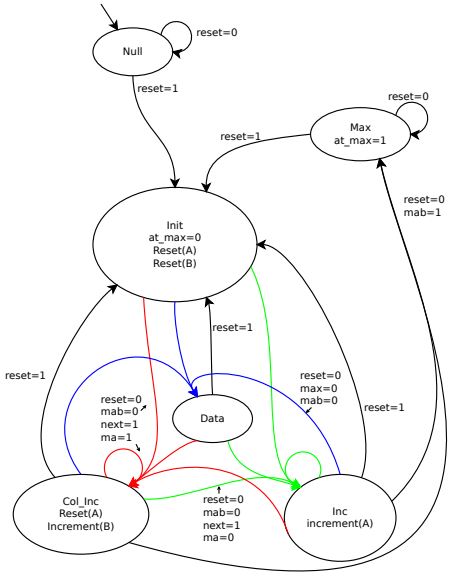

Figure 9: State Diagram for the window generator code, Moore machine: signals `max=c_cuis=max_itr`, `leap=iter_count=v_leap`. Omit unused signals (X) for compactness.

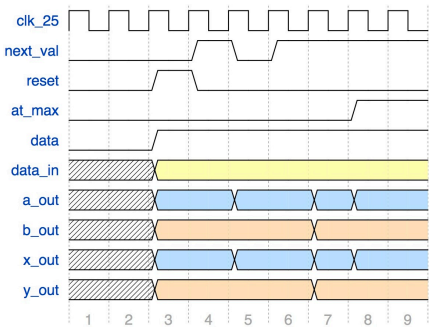

Figure 10: Timing diagram of interface between the window generator and the IFMs.

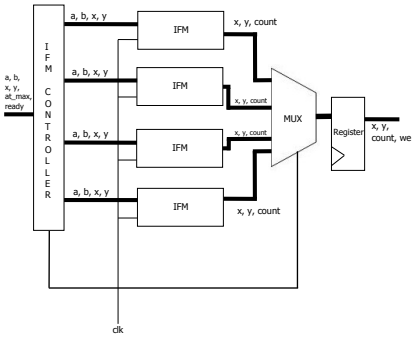

Figure 11: Block diagram for the IFM wrappers

Parallel IFM Control Module

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 control module to resolve these issues.

The IFM control module constantly transmits the (x, y, a, b) tuple currently being expressed by the window generator to each of the IFMs. 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 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.

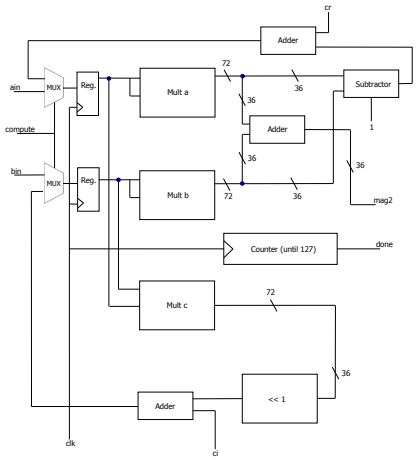

Figure 12: Arithmetic Logic Circuit within each `IFM`

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.

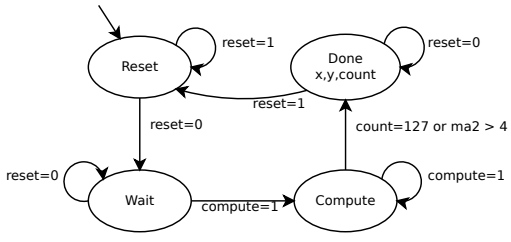

Figure 13: State diagram for a single `IFM`. Moore machine: using abstract transition descriptions. Omits unused signals for compactness.

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.

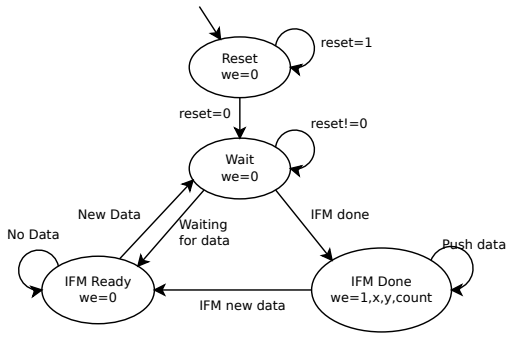

Figure 14: State diagram for a single `IFM` unit (`IFM` handler/wrapper), Moore machine: using abstract transition descriptions.

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 Parallel IFM Control Module as well as the VGA module.

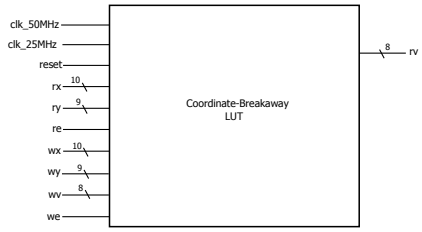

Figure 15: Block Diagram of the Coordinate-Breakaway LUT: signals on the right side are inputs, signals on the left are outputs.

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.

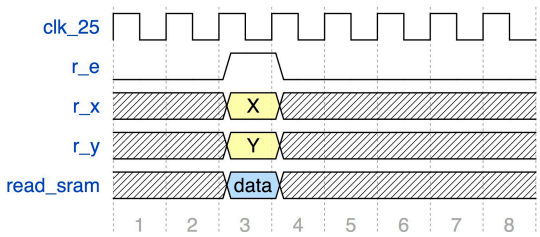

Figure 16: Timing diagram of the interface between the Coordinate-Breakaway LUT and VGA module

Even with every other 25Mhz cycle being dedicated to writing the data being sent out by the IFMs, the SRAM might still miss a coordinate if the IFMs 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.

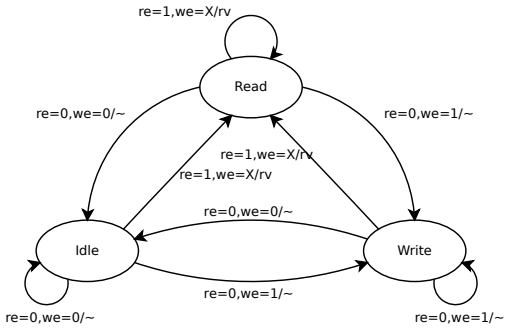

Figure 17: Simplified State Diagram of the Coordinate-Breakaway LUT: Mealy machine, only includes re and we as inputs and rv as an output, with $\sim$ denoting a lack of outputs

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.

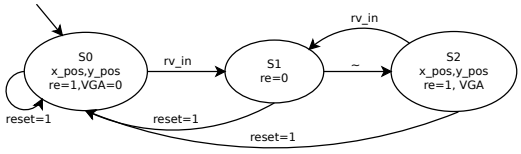

Figure 20: State diagram for the `VGA` module, Mealy machine: $\sim$ stands in for no input/output, and VGA stands in for all the VGA_ signals (`VGA_CLK`, `VGA_HS`, `VGA_VS`, `VGA_BLANK`, `VGA_SYNC`, `VGA_R`, `VGA_G`, `VGA_B`). Omits unused signals for compactness.

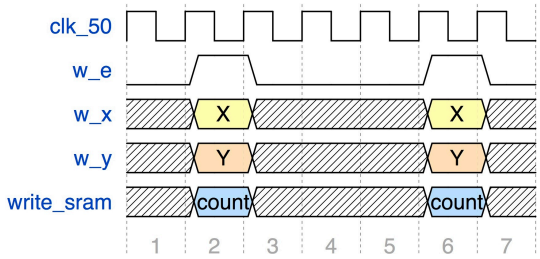

Figure 18: Timing diagram of the interface between the IFMs and the Coordinate-Breakaway LUT.

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.

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.

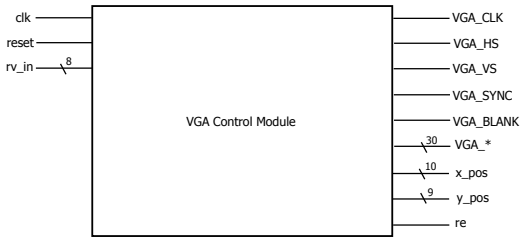

Figure 19: Block diagram of the VGA module

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.

2.3 Parametrization Modules

Software-based Parameterization

Colorization

3 System Performance and Validation

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 10	Feature complete system with accompanying report and presentation.	*As planned.*

5 Challenges and Lessons Learned

6 Reflections and Prospective

All things considered, our team is extraordinarily satisfied with the results that we've achieved. Development was certainly not without its ups and downs, but ultimately, we 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