ECE 385

Fall 2020

Final Project

NIOS-II SoC with Interactive Mandelbrot Set Explorer in SystemVerilog

Maximilian Goldstein – mgg2

ABA / Friday

Andrew Gacek

I. Introduction

For the final project, I designed an interactive FPGA-deployed application which displays fractals and allows the user to pan over different regions of the image at varying degrees of magnification. I extended and adapted the SoC created in Lab 8 to interface with an external USB keyboard and control the movement of a ball displayed over the DE-10's onboard VGA port. Using this project as the starting point, new hardware components were introduced to accomplish the goal of generating a colorized image of the mandelbrot set on an external display and manipulating the image upon receiving certain keycodes from the NIOS-II software component of the design. Because the operations of the VGA and SPI interfaces were discussed more in-depth in previous lab reports, they will not be covered in the discussion that follows. Similarly, the SoC used is nearly identical to that from Lab 8 so I will not go into too much detail describing its configuration and functionality in the overall design. Aside from the major components of the project which will be described next, hardware modules were implemented to keep track of the current state of the fractal. These modules were instantiated in the top-level of the design. Within these modules, a set of registers keeps track of the current X and Y displacement values while another register keeps track of the magnification coefficient. These modules receive keycode signals generated by the NIOS-II SoC as inputs, and output signals to the Fractal Interface module to manipulate the displayed image. Also, a module found online[1] implements the PLL logic needed to derive a 150 MHz clock from the main 50 MHz clock for use in the floating-point calculations. The 150 MHz clock is referred to as the math clock, the 50 MHz clock is referred to as the main clock, and the 25 MHz clock used by the VGA interface is referred to as the frame clock in the source code and discussion.

There were multiple different hardware components needed to implement the final product. Firstly, a set of complex-valued floating-point arithmetic modules were needed to perform the computations involved in generating the fractal and scaling the output image. I used a Floating-Point Unit (FPU) core made available online[2] as the underlying implementation logic for the complex-valued FPU modules tasked with performing precision calculations. The complex FPU modules implemented the arithmetic operations of addition, multiplication, and modulus squared (magnitude squared) with complex-valued operands. Also, modules for converting 64-bit signals between IEEE-754 floating point representation and integer representation were used to implement the computational components of the design.

Secondly, a module was needed to translate the VGA draw-X and draw-Y signals into points on the complex plane. To achieve this mapping, the complex numbers were interpreted as vectors with two components. The real component of the complex values corresponds to the draw-X signal from the VGA controller and the imaginary component corresponds to the draw-Y signal. This interpretation made mapping points on the complex-plane to pixels on the VGA display easier to realize in hardware because transformations between the two planes could be achieved using a linear projection from the VGA coordinate system to the complex coordinate system. Instead of mapping points directly from one space to the other, a projection function was needed to apply horizontal and vertical displacement terms to the center pixel's coordinates and apply the magnification coefficient, which is maintained by the 64-bit magnification register, to produce the effects of moving around the fractal and zooming in and out of the image. Also, the projection function helped define the region over which computations were performed

[1] https://github.com/rkrajnc/minsoc/blob/master/rtl/verilog/altera_pll.v
[2] https://github.com/dawsonjon/fpu/

such that resources were not spent computing over regions outside of view. Also, a module was needed to generate the body of the fractal and assign each point within view an escape value which determines the RGB color encoding of its corresponding pixel. In the design, the hardware component responsible for generating the fractal and computing escape values was separated from the component for assigning color channel encodings because these tasks ultimately relied on different timing and memory constraints. The module which generates the fractal, coupled with the module which generates the complex-plane, are collectively referred to as the Fractal Engine component of the design. The Fractal Engine implements the logic for the mandelbrot algorithm and the control logic for rendering a new fractal whenever the coordinates of the center pixel or magnification coefficients are updated. To generate the fractal, the recursive polynomial $Z_{n+1} = Z_n^2 + c$, with base case $Z_0 = 0$, was evaluated iteratively for each complex-valued point c currently in view. If the squared modulus of Z_n, $|Z_n|^2$ stabilized (converged to some value) after a number of iterations, the point c receives this number of iterations as its escape value. Otherwise, if $|Z_n|^2$ exceeds some prespecified quantity or fails to stabilize after some hard-coded threshold number of iterations, the point c receives an escape value to reflect this and colorize the associated pixel. Visually, the set of points which did not converge make up the dark regions of the fractal image while the points which did converge produce colorful geometric patterns.

Thirdly, a design component was needed to implement the VGA's frame-buffer which would load and store pixel escape values to and from an on-board RAM, as well as assign RGB values to each pixel during a VGA frame interval. This design component instantiates the modules needed to achieve this functionality on its datapath, which includes a dual-port RAM with 640*480 memory addresses corresponding to each pixel and 4-bits of memory per pixel, a pixel colorizer unit which encodes the R, G, and B color channels sent to the DE-10's DAC, address calculation units, and other hardware modules to buffer output values and prevent invalid memory accesses. This design component was particularly challenging to implement because it needed to meet the timing constraints of the VGA controller while also polling the escape value for a given pixel from the Fractal Engine and performing memory operations to store or load the value to or from memory. If the critical path of the Fractal Engine was too long, the VGA image appears distorted or not at all because pixels would not yet be assigned color values while drawing the current frame.

Finally, a design component was needed to instantiate each of the components described above and coordinate their actions in response to signals from the VGA controller and signals from the location and magnification registers. This design component is referred to as the Fractal Core Interface. The modules on its datapath are the VGA pixel frame-buffer module and the Fractal Engine module. The control signals are generated by its control unit, which serves to coordinate the generation of the complex-plane and mandelbrot set about a given point and loading results into the frame-buffer's RAM as they become available.

II. Written Description of the Fractal Engine Component.

The Fractal Engine Component of the project was composed of two subcomponents, as described previously. The first subcomponent generates the complex-plane and applies the appropriate magnification coefficient and translational offsets to the region over which the fractal computations are performed. To convert a VGA pixel coordinate of the form (x,y) to a complex coordinate of the form (re(c), im(c)), which is then used as the input to the fractal generator unit, a linear projection from one space onto the other was performed. To derive this projection, a transformation matrix to fit 640x480 VGA input coordinates onto the complex-plane region where calculations are to occur was used. This

matrix is based on the projection matrix used by OpenGL to apply perspective transformations to an image[3].

Let S_x and S_y denote the magnification coefficients for the x- and y- components of the output, respectively. Let variable n denote the magnification level. Further, the value of the magnification coefficient after n magnifications is computed by raising the magnification coefficient to the power of n. For example, n = 2 yields $S_x^2 = S_xS_x$, n = 3 yields $S_x^3 = S_x^2S_x$, and so on. Intuitively, if the factor by which the magnification coefficients increase with the magnification level was 0.90, then after 3 magnifications, the magnification coefficient would be $(0.90)^3 = 0.73$. That is, the output image would be scaled to cover about 73% of the region it covered initially over the same 640x480 display region. The process for demagnifying is similar except the factor for decreasing magnification coefficients is 1.10. Thus, the output image would be scaled to cover about 133% of the region it covered initially after 3 de-magnifications. Keycodes corresponding to 'Q' from the NIOS-II triggered a decrease to the magnification coefficient (zoom-in operation), and keycodes corresponding to 'E' increased the coefficient (zoom-out operation). The values of S_x^n and S_y^n are maintained by a magnification register module in the top-level of the design. The region on the complex-plane over which the mandelbrot fractal occurs is described by the set of points c = x + iy, where the values of the tuple (x,y) are represented by the set[4]

$$\{ (x,y) \mid x \in [-2, 2],\ y \in [-1, 1] \}.$$

That is, the range of values in the x-dimension has size 4 and the range of values in the y-dimension has size 2 [4]. To map the VGA coordinate space into the 4x2 drawing region of the fractal on the complex plane, the horizontal and vertical components of input tuple need to be proportioned according to the dimensions of the mandelbrot region and scaled by the current magnification coefficients. This matrix, M, is referred to as the perspective transformation matrix[3].

$$M = \begin{bmatrix} (S_x)^n/(640/4) & 0 & 0 \\ 0 & (S_y)^n/(480/2) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Next, transformations are needed to translate the center pixel coordinate of the VGA display area to the center of the fractal plane drawing region. The center of the VGA display is the pixel described by tuple (320, 240). Then, to translate the center pixel coordinates on the VGA plane by some horizontal and vertical offsets, denoted δx and δy, the linear transformation Φ is applied where

$$\Phi = \begin{bmatrix} 1 & 0 & \delta x \\ 0 & 1 & \delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 320 \\ 0 & 1 & 240 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \delta x + 320 \\ 0 & 1 & \delta y + 240 \\ 0 & 0 & 1 \end{bmatrix}.$$

Then, for a given coordinate on the VGA display located at (x,y), the values of x and y are subtracted from the quantities $\delta x + 320$ and $\delta y + 240$, respectively, because the pixel at coordinates (0,0) is located at the top-left of the VGA display and x increases from left to right while y increases from top to bottom. In the implementation, the x- and y- coordinates vary with the draw-X and draw-Y signals generated by the VGA controller, and δx and δy values change each time a direction key signal is generated by the SoC and updates the associated value in the position register module. This transformation is interpreted as increasing δy shifts the center of the displayed image downward and increasing δx shifts the displayed

[3] https://programmersought.com/article/44781734944/
[4] http://physics.ucsc.edu/~peter/115/mandelbrot.pdf

image to the right. On the other hand, increasing the draw-Y coordinate maps pixels onto a higher region of the fractal-plane while increasing the draw-X coordinate maps pixels onto a more-leftward region. Decreases to the input draw-coordinates produce an opposite effect. These effects were realized in the implementation by having the Up and Left direction keys ('W', 'A') decrease the current δy and δx values, respectively, and mapping the Down and Right direction keys ('S', 'D') to increases in the current δy and δx values, respectively. The resulting matrix, denoted Φ' is described as

$$\Phi' = \begin{bmatrix} 1 & 0 & \delta x + 320 \\ 0 & 1 & \delta y + 240 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -x \\ -y \\ 1 \end{bmatrix} = \begin{bmatrix} \delta x + 320 - x \\ \delta y + 240 - y \\ 1 \end{bmatrix}$$

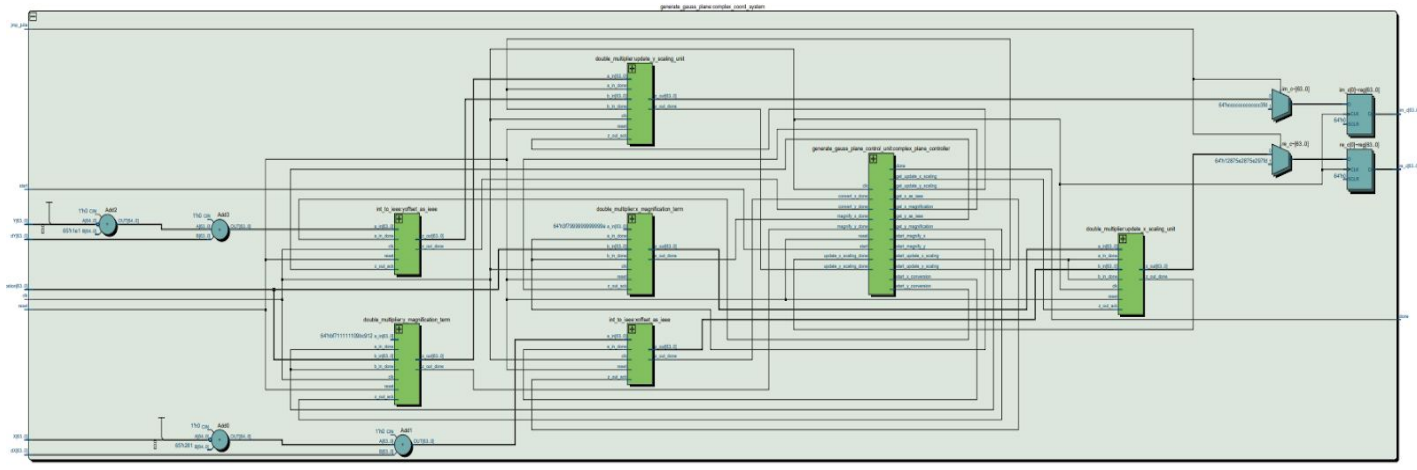
Finally, the resulting transformation matrix β to convert VGA coordinates of the form (x,y) to complex coordinates of the form (re(c), im(c)) is obtained by multiplying M and Φ' to obtain $\beta = M\Phi'$. That is, β equals

$$\begin{bmatrix} (S_x)^n/(160) & 0 & 0 \\ 0 & (S_y)^n/(240) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \delta x + 320 - x \\ \delta y + 240 - y \\ 1 \end{bmatrix} = \begin{bmatrix} ((S_x)^n/(160))(\delta x + 320 - x) & 0 & 0 \\ 0 & ((S_y)^n/(240))(\delta y + 240 - y) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

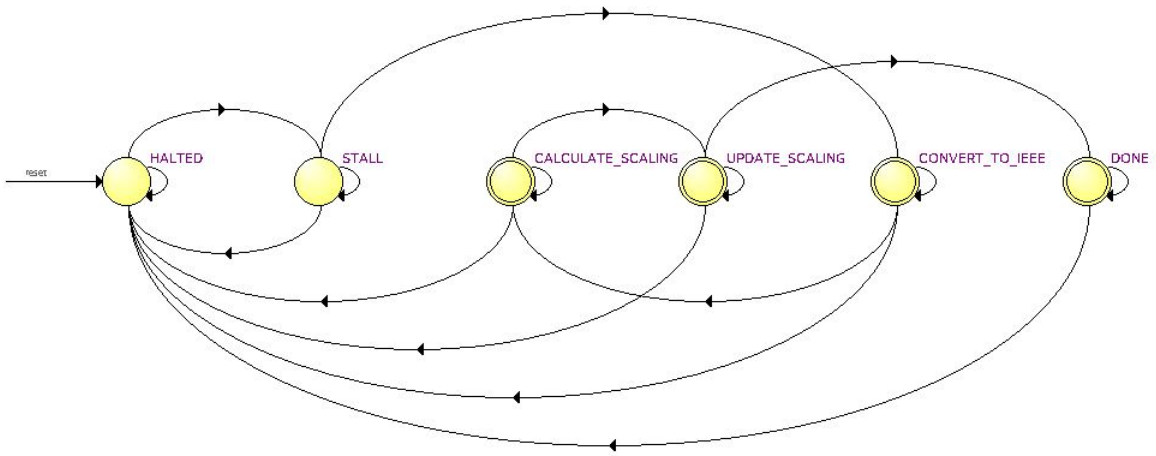
Finally, the general result is obtained that for VGA coordinates of the form (x,y), translational offset values (δx, δy) , and magnification coefficients S_x^n *and* S_y^n , the corresponding points on the fractal plane are

$$\text{re(c)} = \frac{S_x^n}{160}(\delta x + 320 - x)$$
$$\text{im(c)} = \frac{S_y^n}{240}(\delta y + 240 - y)$$

The module implementing this mapping routine first calculates the x- and y- offset values for this point on the complex plane using input translation terms δx and δy and the relations derived earlier and shown nested inside the parentheses in the equations describing re(c) and im(c). Once calculated, these offset values are converted into 64-bit floating point representation using one of the provided FPU cores found online. Then, the magnification coefficients of the x- and y- coordinates are computed using the floating-point multiplier unit found online to obtain values for $S_x^n/160$ and $S_y^n/240$ at magnification-level n. Lastly, these two quantities are multiplied together using another floating-point multiplier to obtain output values for re(c) and im(c) at the input coordinate (x,y). The RTL diagram and state machine for this subcomponent are shown below.



RTL view of generate_gauss_plane module.



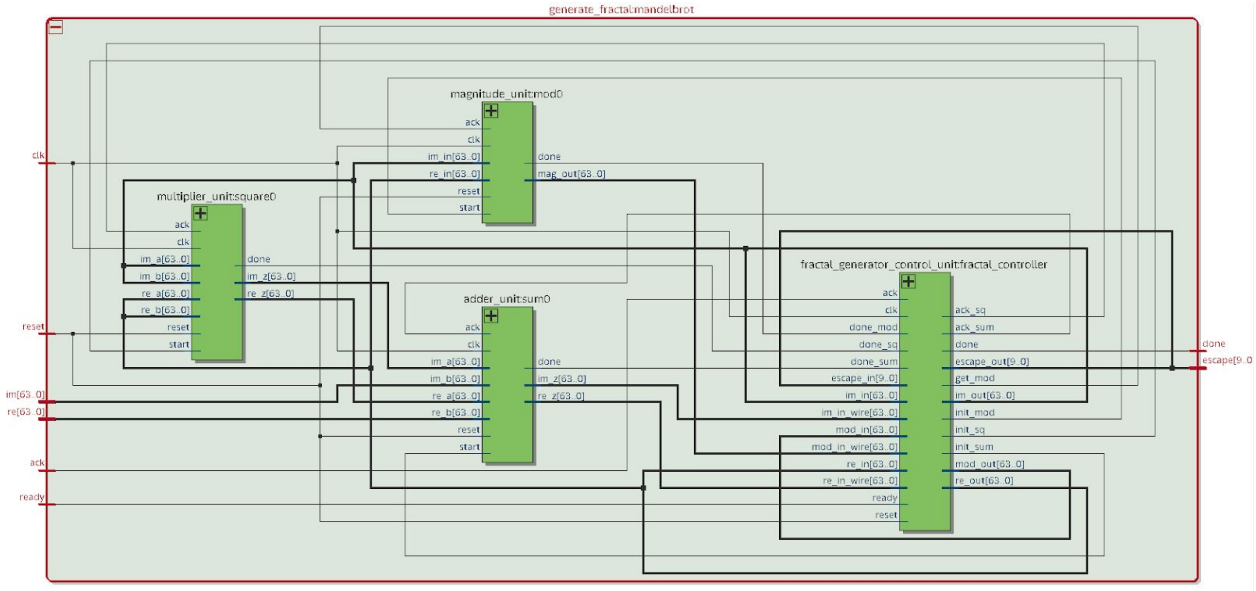
State machine for generate_gauss_plane module.

The first state is HALTED, triggered by the rising edge of the reset signal. In HALTED, the output values for re(c) and im(c) are set to zero. If reset is low, the state transitions to the STALL state where it remains until the next rising edge of the input signal named start. The start signal is generated by the Fractal Engine control unit and invokes this subcomponent to begin the next computation cycle. The first transition out of the STALL state is to the state named CONVERT_TO_IEEE in which the horizontal and vertical offset terms are converted into IEEE-754 floating point representation, as described previously. Once the conversion module sends the done signals for both x- and y- terms, the next state transition is to the CALCULATE_SCALING state. Here, the current magnification coefficient is multiplied by a magnification factor to compute the next magnification coefficient. Once the done signal is received from the double multiplier implementing this calculation, the next state transition is to the UPDATE_SCALING state. Here, the horizontal and vertical offset terms yielded during the CONVERT_TO_IEEE state are multiplied by the magnification coefficient just computed to obtain values for re(c) and im(c). After both multiplier units implementing these calculations send their done signals, the state machine transitions to the DONE state where acknowledgement signals are sent to each compute module on the datapath and the done signal is sent to the Fractal Engine to invoke a transition in its state machine. The state controller for the generate_gauss_plane module remains in the DONE state until the next falling edge of the start signal, where it transitions to the HALTED state and begins the next cycle.

After the corresponding values of (re(c), im(c)) are computed for a given input coordinate (x,y), the complex-point c is fed as input to the other subcomponent on the Fractal Engine's datapath, the generate_fractal module. This module implements the Mandelbrot algorithm by recursively evaluating the polynomial $Z_{n+1} = Z_n^2 + c$ about the input c until the squared magnitude of Z_n exceeds the value 2 or the integer n exceeds the threshold value of 128. These threshold values were chosen based on readings found online about plotting the mandelbrot set[5] [6]. The squared magnitude was evaluated rather than the first-order magnitude to preserve resources and avoid the need for an additional square-root FPU module. In the implementation, this choice produces no change to the output image because the squared magnitude is checked for stability or convergence between iterations rather than taking on specific values. As long as

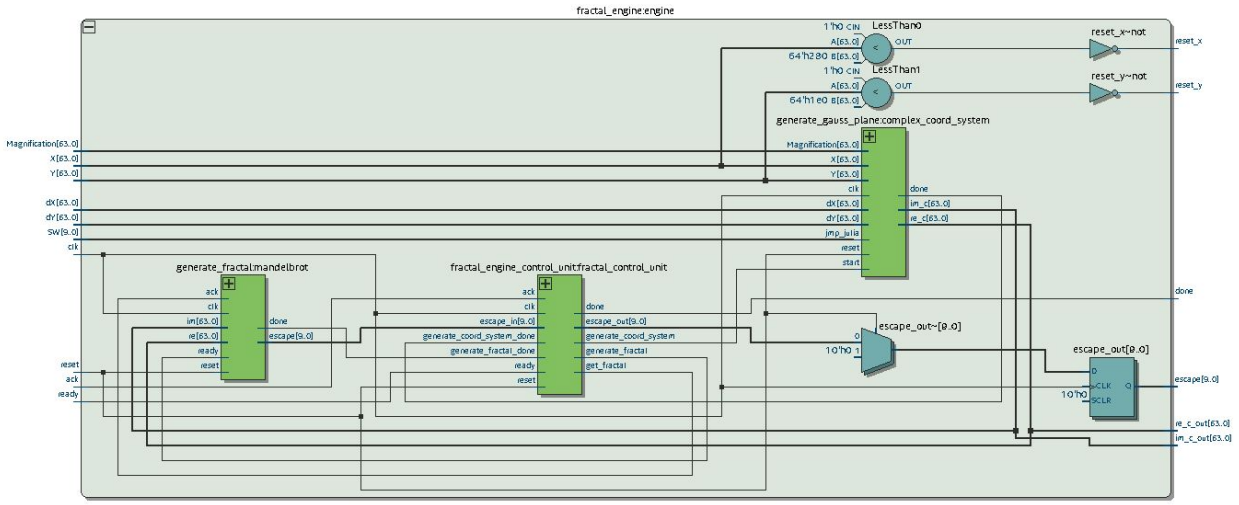
[5] https://www.math.univ-toulouse.fr/~cheritat/wiki-draw/index.php/Mandelbrot_set
[6] http://paulbourke.net/fractals/mandelbrot/Ruben_van_Nieuwpoort.html

the squared magnitude changes between iterations, the control unit transitions states accordingly. The module computes the escape value associated with a point in three distinct phases. During the first phase, the squared magnitude of the complex-point described by (re(c), im(c)) is calculated and compared against the threshold value of 2. During the next phase, the values of the real- and imaginary- components are squared using a complex multiplier unit and the result is stored in an internal register. Then, during the next phase, the sum of the squared complex-components and the corresponding input components is computed and stored in an internal register. If the component values of the next iteration equal the values from the previous iteration, then the result has stabilized and the point receives an escape value equal to the maximum threshold number of iterations. This result is interpreted as the squared magnitude at the complex-point c blowing up to infinity after many iterations. Otherwise, the escape value is incremented by one and the process repeats at phase 1 after updating the current values of re(c) and im(c) with the next values, respectively. When the component values in phase 3 are changing between iterations, but the squared magnitude exceeds the threshold value of 2, the point receives an escape value corresponding to the current number of iterations. Then, another module in the design colors the pixel corresponding to this point using the escape-value to index into a lookup table of RGB color values. The RTL diagram for this module is shown below.



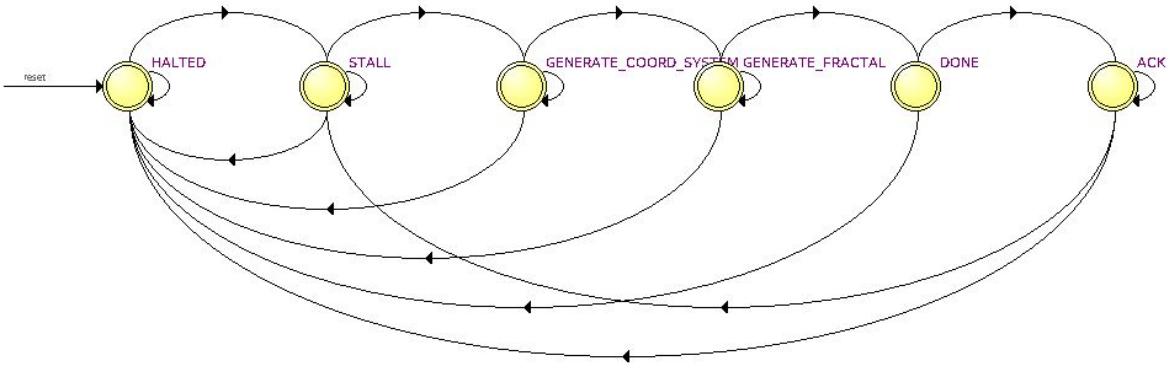
RTL view of generate_fractal module.

The overall RTL view of the Fractal Engine component of the design is shown in the diagram below. Here, one can see the connections between submodules of the Fractal Engine with each other and connections between submodules and signals generated by the Fractal Engine's control unit. The outgoing signals reset_x and reset_y compare the current x- and y- input coordinates with the dimensions of the VGA display. If x exceeds 640, reset_x is set to 1 and the Fractal Core Interface component begins to draw the next row. Similarly, if y exceeds 480, reset_y is set to 1 and the Fractal Core Interface component begins drawing the next frame. Otherwise these signals remain low, respectively.



RTL view of fractal_engine module

Additionally, the state machine for this component is shown below.



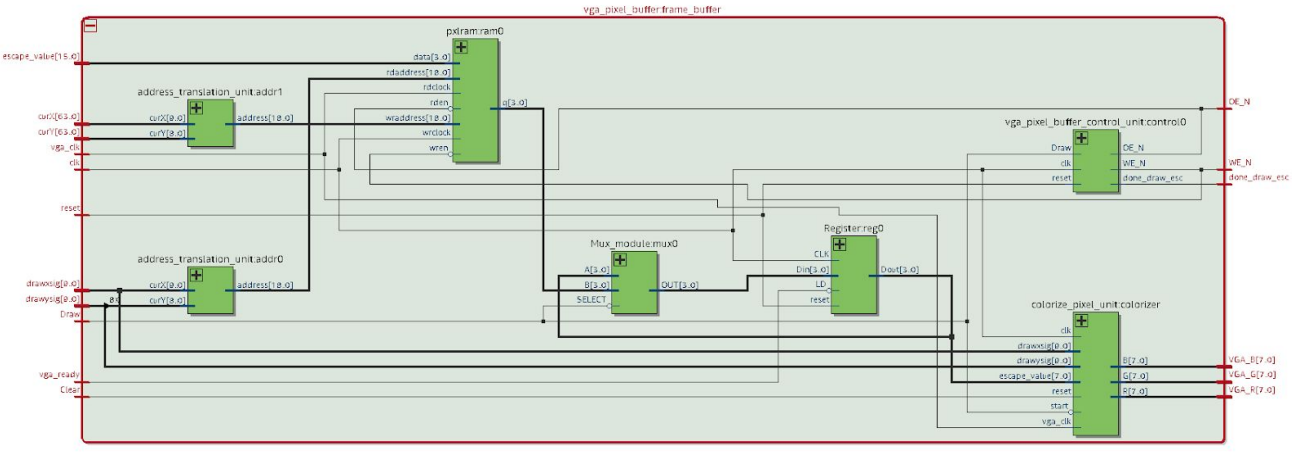
State machine for fractal_engine module

The HALTED state is entered on the rising edge of the reset signal. In this state, the outputs of all subcomponents on the datapath are cleared. Then, if the reset signal is low, the next state is STALL in which the next escape value is set to 0 and the control unit remains in this state until the input signal ready goes high. Then, the next state is GENERATE_COORD_SYSTEM in which the generate_gauss_plane module is prompted to begin calculating values of c corresponding to a pair of input coordinates. Once the done signal is generated from the generate_gauss_plane module, the next state is GENERATE_FRACTAL in which the generate_fractal module is invoked to begin calculating the escape value associated with a given point c on the complex-plane. When the generate_fractal module completes and sends its done signal, the next state is DONE in which the output value for escape is updated with the result from the generate_fractal module and the controller transitions to the ACK state. In ACK, the acknowledgement control signals are sent to the modules on the datapath as well as the higher-level Fractal Interface module which instantiated the Fractal Engine module.

III. Written Description of the VGA Frame Buffer Component.

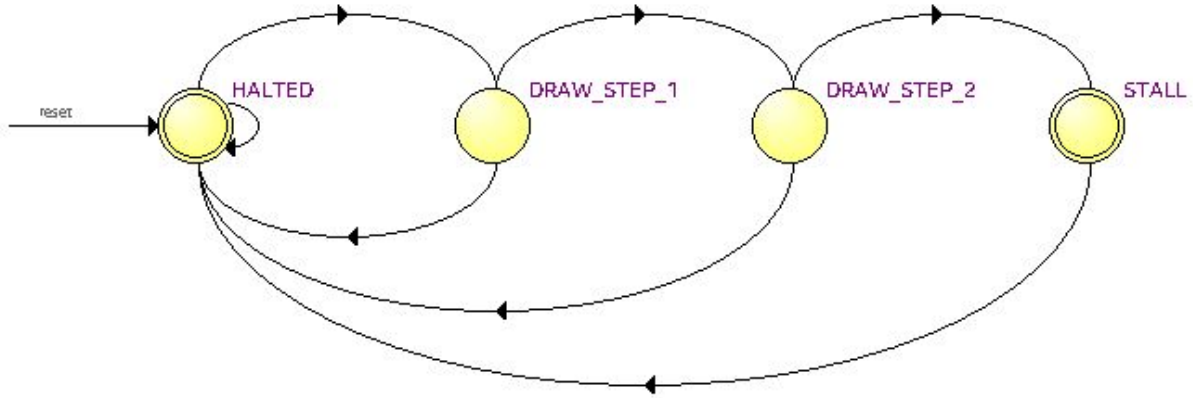
The next major design component of the project was the VGA frame-buffer and memory interface. This design component was responsible for interfacing with the VGA controller in the top-level unit of the design and writing and reading escape values to and from memory addresses corresponding to the current pixel being iterated over and displayed. Once an escape value has been read from memory, the

corresponding entry in a lookup table of RGB encodings is obtained and the channels for each color component are updated to reflect the escape value. Ultimately, the escape values determine the color assigned to a pixel during the next frame. Each time the fractal is regenerated or reset, the contents of memory are updated with new values. A dual-port RAM is needed to implement the frame buffer because data may be written or read to and from memory at different speeds. The VGA controller reads memory at the frame clock speed of 25 MHz, while the results are written to memory at the speed of the Fractal Engine's control unit which runs at 50 MHz. Because I wanted to avoid the use of tri-state buffers in my design, I opted for a design approach similar to the mem2IO module from lab6 to implement the memory interface. In my design, I use WE_N and OE_N signals to control when the contents of memory can be accessed and modified. I also use a multiplexer and register to select the source of data to use to colorize a pixel and store the current data for reuse in the event the data is not available when the VGA draw signals request it. The combination of WE_N and OE_N signals, ready signals from the VGA, and ready signals from the Fractal Engine, help select a proper data source for coloring a pixel and prevent invalid memory accesses from occurring. In implementation, a feedback loop is created between the output of the register and an input line to the multiplexer such that output values are reused when the results from memory are not ready or updated otherwise. Then, the output of the multiplexer is fed into the register and the output of the register is fed into a pixel color mapping module whose mapping mechanism uses a lookup table based on the escape value as described. One drawback of this design implementation is that each pixel location is associated with a memory address, limiting the number of bits that can be associated with each address. With 640*480=307200 different addresses, the implementation was limited to 4-bits per address. In effect, this limits the width of the escape values stored in memory to 4-bits and the color palette to 16 unique entries for each escape value. Although limited to 16 colors, the output image was still able to render finer details within different regions and geometric patterns within the fractal. Address translation units were used to convert input coordinates of the form (x,y) into memory read and write addresses within the RAM. The translation is straightforward and calculates addresses in row-major order similar to how one accesses entries in a flattened two-dimensional array in C. An input signal named Draw is used to determine whether the frame-buffer is reading or writing a value to or from memory. When Draw is high, the escape value already loaded into memory is used to assign the color values to the pixel. Otherwise, the escape value calculated during the last iteration of the Fractal Engine is loaded into memory and colorizes the pixel. Shown below is the RTL view of the vga_pixel_buffer module used to implement the frame-buffer and color assignment module. The output signal done_draw_esc is routed back to the Fractal Interface Core's control unit to coordinate the actions of modules on its datapath.



RTL view of vga_pixel_buffer module

The state machine for this component is shown below.

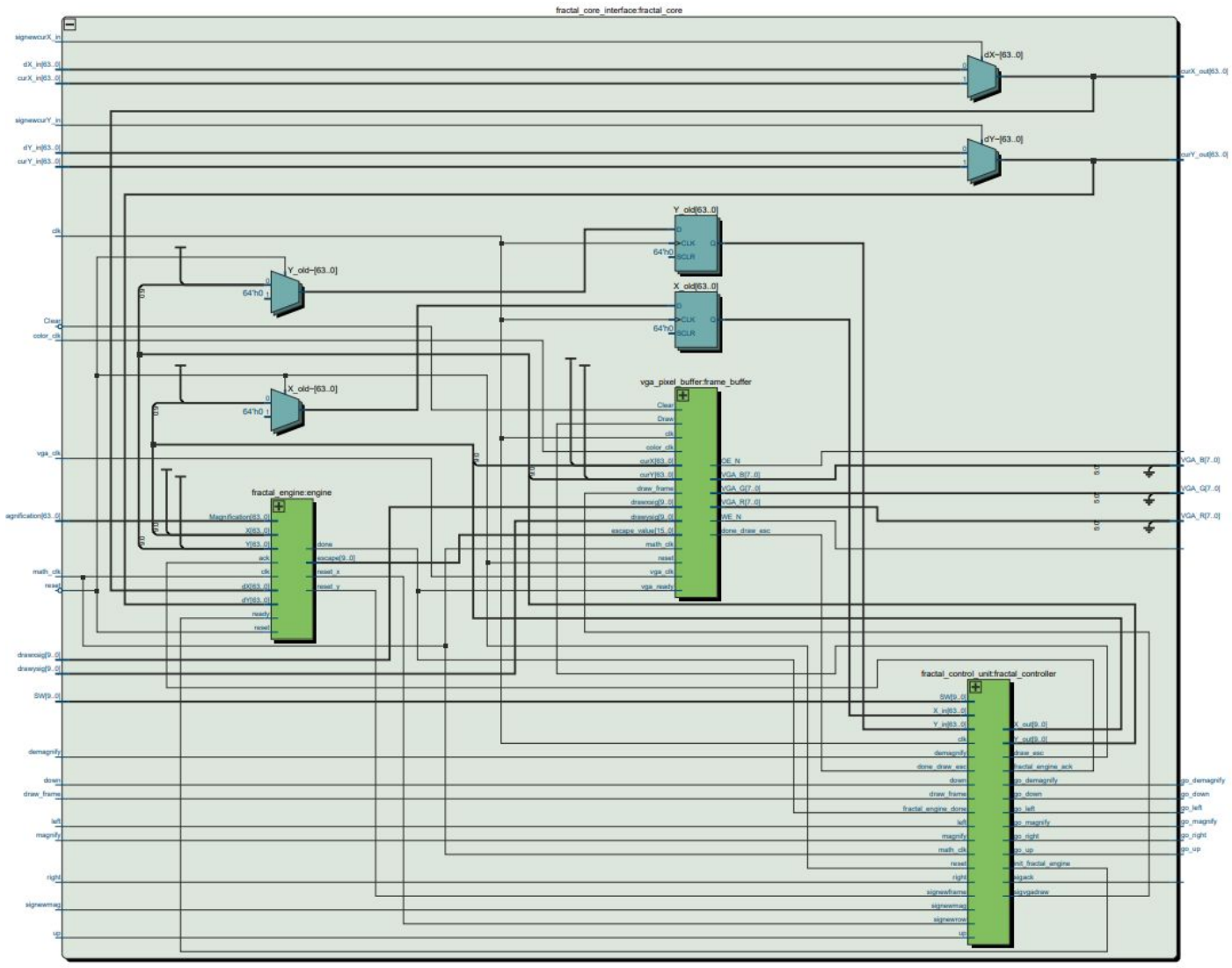


State machine of vga_pixel_buffer module

The first state is HALTED which is triggered by a reset signal. In HALTED, the done_draw_esc signal is raised and WE_N is set to 1. The transition out of HALTED occurs when the input signal Draw is raised. The next state is DRAW_STEP_1, where WE_N is set to 0 and the transition to DRAW_STEP_2 occurs in the next clock cycle. The draw states span multiple clock cycles because the memory may need multiple cycles before the contents of memory are able to be written to or read from. In DRAW_STEP_2, WE_N remains low and the next state is STALL. In STALL, WE_N is set to 1 and the next state is HALTED where the process repeats again each time the Draw signal is raised.

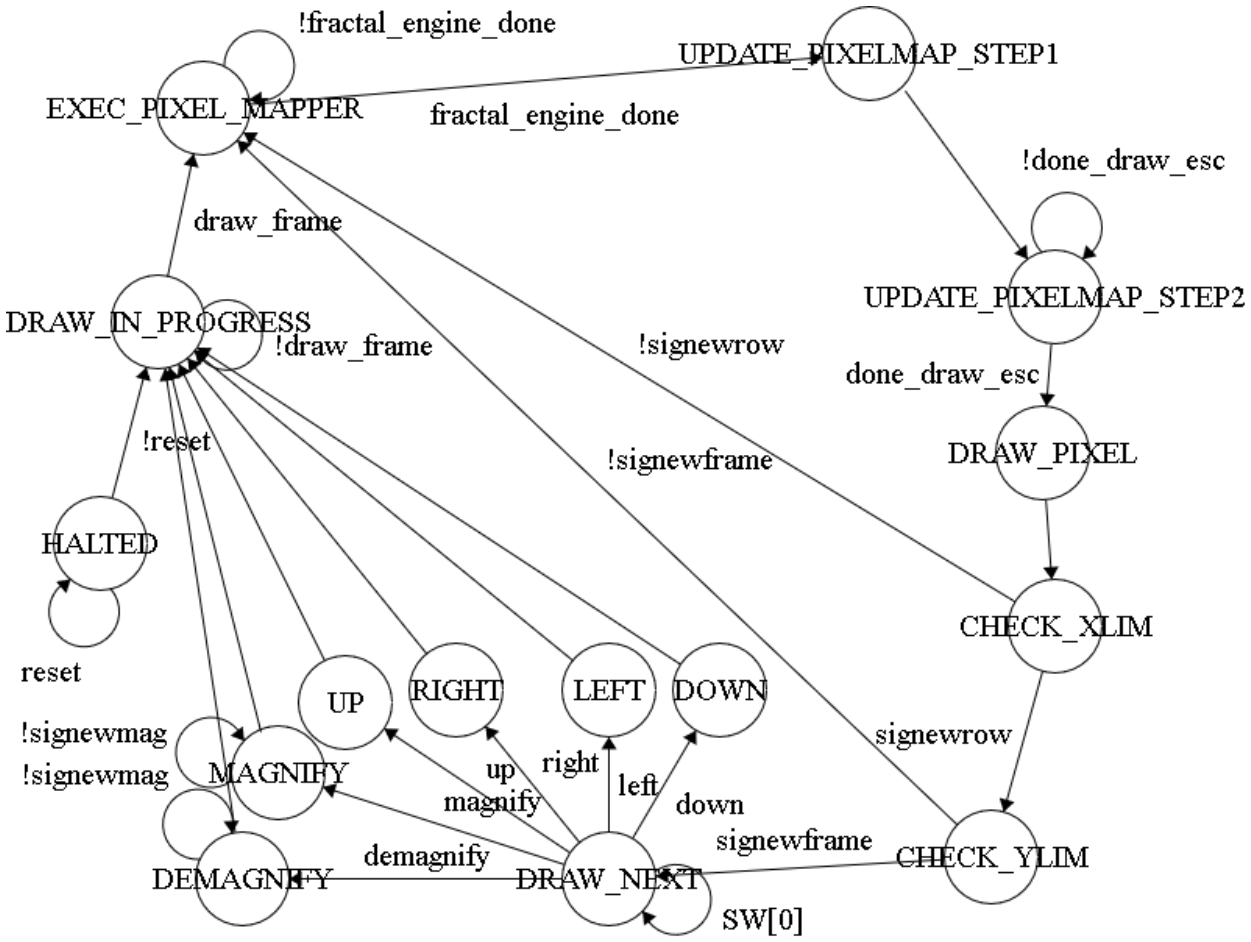
IV. Written Description of the Fractal Interface Component.

The next component of the design, and the component instantiated in the top-level is the Fractal Interface Component module. This module instantiates the Fractal Engine component and the VGA frame buffer component and coordinates the actions of the two components with respect to their different timing constraints, generated output signals, and with respect to signals received from the NIOS-II software component of the design. Upon receiving signals to move up, left, down, or right, from the top-level module, the Fractal Interface component updates the current values of the x- and y- displacement registers and prompts the Fractal Engine to begin rendering a new fractal centered around the new point. Similarly, upon receiving the magnify or demagnify signals, this module prompts the Fractal Engine to redraw the fractal at a higher or lower resolution, respectively. In addition to controlling the rendering of fractals, this module oversees and facilitates the exchange of acknowledgement signals between the various components of the design. Once an escape value for a pixel has been calculated by the Fractal Engine, the Fractal Engine generates its done signal and the Fractal Interface connects this signal to the VGA frame-buffer to begin the process of storing this value in memory and encoding the color channels for the DAC on the DE-10 so the appropriate color value is drawn to the pixel during that frame clock interval. Different control signals are needed to track the progression of both calculations and drawing to the output display. The Fractal Interface's control unit manages both types of signals and implements the transition logic between states required for performing calculations about a central coordinate and displaying the results to a VGA display. The RTL view showing the connections between subcomponents of the Fractal Interface Component is shown below.



RTL view of Fractal Interface Core component of the design.

The state machine for the control unit of the Fractal Interface component of the design is shown below. The default transition from the DRAW_NEXT state to DRAW_IN_PROGRESS state when no positional or magnification keypresses are received is not shown for the sake of reducing clutter. Similarly, the transition labels from MAGNIFY and DEMAGNIFY states to DRAW_IN_PROGRESS are not shown but the transition occurs when the input signewmag is set to 1.



State diagram for Fractal Interface Core module of the design.

In the HALTED state, which is entered on the rising edge of the reset signal, the control signals sigxnext and sigynext are cleared and the signal sigvgadraw is set to 1. If reset is low, the controller transitions to state DRAW_IN_PROGRESS in which sigvgadraw is set to 0 and sigack is raised. If input signal draw_frame is raised, the controller transitions to state EXEC_PIXEL_MAPPER where the control signal init_fractal_engine is raised, signaling the Fractal Engine component of the design to begin its next execution cycle. When the fractal_engine_done signal is received from the Fractal Engine component, the controller transitions to the state UPDATE_PIXELMAP_STEP_1 in which control signal draw_esc is set to 1 to signal to the VGA frame-buffer component of the design to load the escape value into memory and colorize the corresponding pixel. The next state is UPDATE_PIXELMAP_STEP_2 which maintains the value of draw_esc for another clock cycle since the memory read and write operations and color assignments may take multiple clock cycles. From UPDATE_PIXELMAP_STEP_2, if done_draw_esc signal is received from the VGA frame-buffer module, the controller transitions to the DRAW_PIXEL state in which both draw_esc and sigvgadraw are set to 1. The next state transition is to the CHECK_XLIM state where the fractal_engine_ack signal is raised to indicate to the Fractal Engine module that the pixel mapping for the current x-coordinate value is completed and the value of sigxnext is updated with an incremented x-coordinate value. If input signal signewrow from the Fractal Engine is low, the controller transitions to state EXEC_PIXEL_MAPPER, otherwise it transitions to state CHECK_YLIM in which the value of sigxnext is reset to 0 and the value of sigynext is updated with the next y-coordinate value. Then, if the input signal signewframe from the Fractal Engine is low, the

controller transitions to state EXEC_PIXEL_MAPPER, otherwise it transitions to state DRAW_NEXT to begin drawing the next frame. In DRAW_NEXT, sigxnext, sigynext, and sigvgadraw are set to 0. The default next-state transition is to DRAW_IN_PROGRESS, however if the input signals for up, down, left, right, magnify, or de-magnify are received, the state controller transitions to the respective state and raises the control signals to apply the desired changes to either the position registers or magnification register, respectively. However, in the MAGNIFY and DEMAGNIFY states, the state controller does not transition to the DRAW_IN_PROGRESS state until the signewmag input signal is received from the magnification register module in the top-level module of the design. Additionally, if SW[0] is set, the controller remains in state DRAW_NEXT and does not draw the next frame.

V. Written Descriptions of Complex Floating-Point Modules, Position Register Module, and Magnification Register Module

The Complex Floating-Point modules, referred to collectively as the CFPU core, were implemented using the FPU modules found online[2]. Instead of using the Altera FPU modules that come with Quartus and are created using Megafunctions, I determined it was advantageous to use the implementation found online because the Altera FPU modules were not configurable to include output signals indicating the computation was complete or input signals for acknowledging results and starting computation cycles. The Altera FPU modules relied on the designer to implement state machines which tracked the number of clock cycles since the computation began to determine whether results were ready rather than loading the results upon receiving an acknowledgement or done signal from the FPU module. I found that designing state controllers proved to be more challenging and error-prone when using Altera FPU modules because results were implicitly inferred to be ready after a certain number of clock cycles rather than explicitly made ready by a generated signal within the module. This made coordinating the timing of different modules difficult because different components of the design performed operations at different speeds and modifications to the clock frequencies driving the computation components of the design significantly altered critical paths for various signals in each design component. While possible to have implemented the design using Altera FPU modules, it was much easier to use the modules made available online because of the additional ports they included.

The CFPU Addition unit was implemented straightforwardly by computing the sum of the real- and imaginary- terms of the inputs component-wise and outputting each respective sum as the real- and imaginary- component of the result. That is, for inputs $z_1 = a_1 + ib_1$ and $z_2 = a_2 + ib_2$, the output of a complex addition operation would be $(a_1+a_2) + i(b_1+b_2)$.

The CFPU Multiplication unit was implemented by exploiting the identity that the product of two complex numbers, $z_1 = (a + bi)$ and $z_2 = (c + di)$, equals $(ac - bd) + i(ad + bc)$. The complex-valued multiplication module uses four floating-point multipliers and two floating-point adders to compute the complex product. The complex multiplication algorithm implemented by the state machine calculates the products of the terms ac, bd, ad, and bc, in one state and transitions to the next state once the done signals are received from each respective FPU multiplication unit. Next, the algorithm computes the sums (ac - bd) by ORing the most-significant-bit of the bd-term to negate its value, and (ad+bc), using two FPU addition units. The results from these two modules become the real- and imaginary- components of the output, respectively. The next states of the algorithm implementation acknowledge the result from each FPU module used so the results can be discarded and the module can reset its internal state before the next computation cycle. The implementation of the CFPU multiplication unit generalizes the multiplication of any two complex-valued operands such that the unit can be used to compute the product of two different operands or the square of an operand. This implementation choice would have helped in the design of the

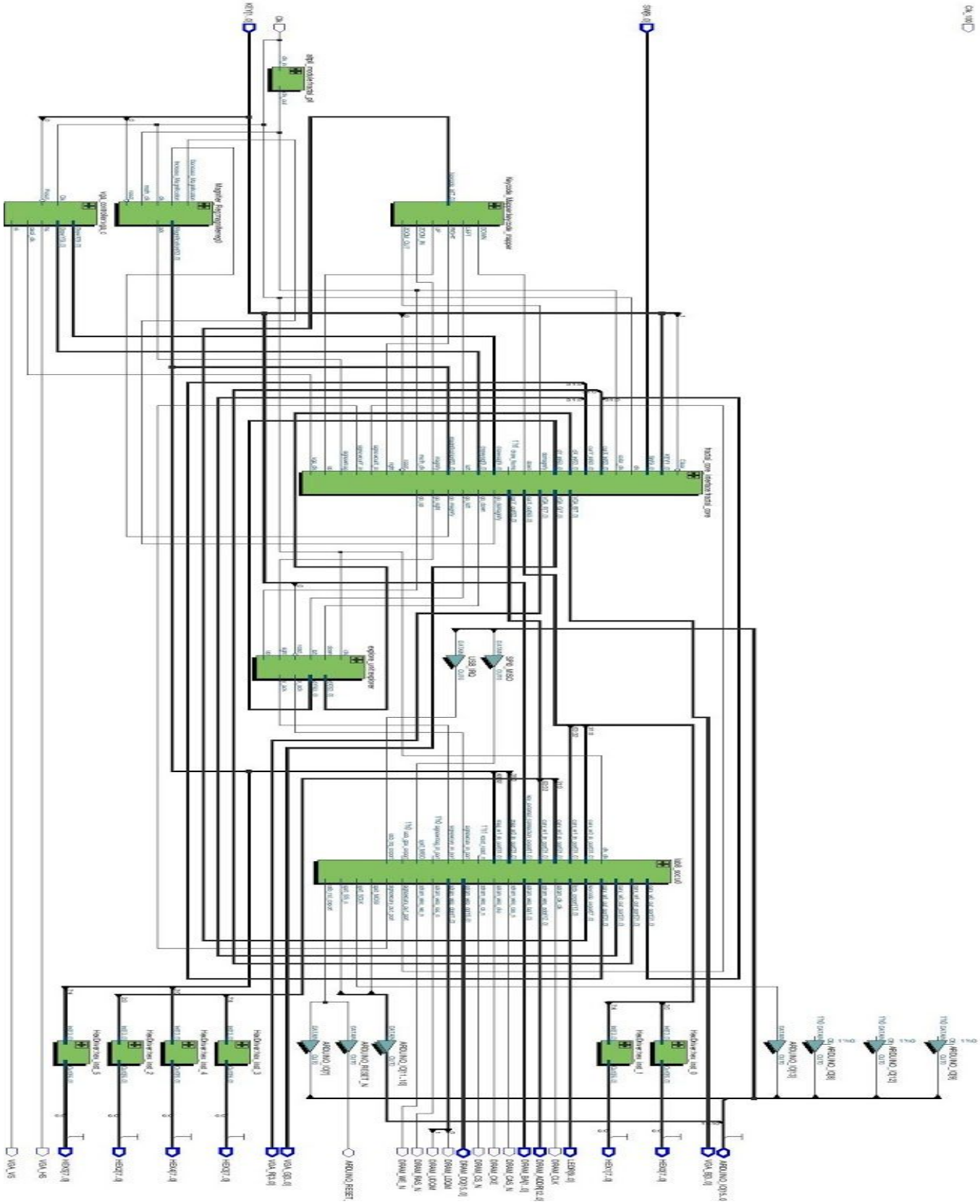
advanced feature to generate generalized Julia set fractals since they are described by polynomials which may or may not include higher-order terms.

The CFPU modulus unit computes the squared modulus (magnitude) of the input operand and outputs a floating-point scalar value representing the length squared of the operand on the complex-plane. Under this interpretation, the operand is represented by a vector and the length is interpreted as the distance between the origin and tip of the vector on a two-dimensional plane. For a given input $z = a + ib$, the output generated by the modulus unit is $|z|^2 = Re(z)^2 + Im(z)^2$, where $Re(z) = a$ and $Im(z) = b$, respectively. The algorithm used to implement the state controller for the module computes the squares of each component of the input in one state, then transitions to the next state where the results are acknowledged upon receiving done signals from the FPU multiplication units used during the calculation. The next-state transition is to the state where the sums of each component are calculated and upon receiving done signals from the FPU addition modules used, the controller transitions to the state where the sums are acknowledged so the FPU modules used can discard the results and prepare for the next computation cycle.

Together, each of these modules, along with modules for converting 64-bit signals to and from IEEE-754 floating point representation and integer representation made available online, made up the set of arithmetic units used for calculations on the complex plane and calculations to manipulate the magnification coefficients.

As discussed in the introduction, the module responsible for maintaining the current values for the x- and y-displacement terms when generating the fractal is instantiated in the top-level of the design. Similarly, the module responsible for maintaining the current magnification coefficient value is instantiated in the top-level of the design. Both of these modules depend on a Keycode_Mapper module which receives the keycode generated by the NIOS-II SoC as an input signal and outputs signals whose values correspond to the type of keypress received. The logic for this module is similar to that implemented in lab8 to change the trajectory of the ball at the direction of signals sent over a USB keyboard to the SoC. The Keycode_Mapper module generates 1-bit signals Left, Right, Up, Down, Magnify, and Demagnify, in the top-level of the design which are sent into the explorer_unit module, Magnifier_Reg module, and Fractal Core Interface module, respectively. The design for the explorer_unit was described previously, but I will briefly add that it is implemented using two submodules, one to track changes to the x-displacement register and one to track changes to the y-displacement register. When a directional keypress is detected, the Fractal Core Interface module raises the control signal corresponding to the keypress and the appropriate displacement register is incremented or decremented by a hard-coded step-size parameter. Similarly, when a magnification modifier keypress is detected, the Fractal Core Interface module sets the control signal associated with that keypress and the magnification coefficient is scaled down by a constant-factor to provide the effect of zooming-in on a region or scaled up by a constant-factor to create the effect of zooming-out over a region. When the magnification operation is complete, the Magnifier_Reg module sends an acknowledgement signal to the Fractal Core Interface module. The RTL view of the top-level of the design is shown below to illustrate the connection between signals generated by the modules of the design.

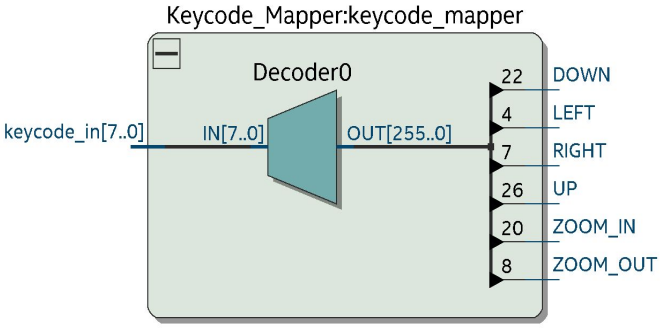
RTL view of top-level design module.



VI. Module Descriptions

Module: lab8.sv
Inputs: Clk, Clk_100, [1: 0] KEY, [9: 0] SW,
Outputs: [9: 0] LEDR, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, HEX5, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [1:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B
InOut: [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N
Description: This module instantiates the SoC needed to perform register keycodes and initialize the external USB interface. Also, this module instantiates the Fractal Core Interface module used to generate the fractal and encode the RGB signals sent to the DAC hardware. Also, instantiates Magnification Registers and Position registers. Generates a 150 MHz clock from Clk input to drive computation units. Implements logical connections between SoC signals, internal state registers, VGA interface, and Fractal Core Interface module.
Purpose: This module represents the top-level module for this lab.

Module: Keycode_Mapper.sv
Inputs: [7:0] keycode_in
Outputs: UP, DOWN, LEFT, RIGHT, ZOOM_IN, ZOOM_OUT
Description: This module receives the keycode generated by the NIOS-II SoC and generates signals corresponding to the type of keypress received.
Purpose: This module decodes keypresses and generates signals used to manipulate viewing perspective and drive circuit operation in a centralized way rather than having each module decode keypresses individually.



RTL block diagram of Keycode Mapper module

Module: Magnifier_Reg.sv
Inputs: clk, math_clk, reset, Increase_Magnification, Decrease_Magnification
Outputs: [63:0] Magnification, ack
Description: Calculates the magnification-coefficient in response to a signal to increase or decrease current magnification level. The output magnification signal is used to render images at different levels of resolution and scaling.
Purpose: Module implements logic for zooming-in or zooming-out on a region by decreasing or increasing the magnification coefficient, respectively. Maintains state of magnification register between keypresses and frames. Used by other modules to determine output scaling.

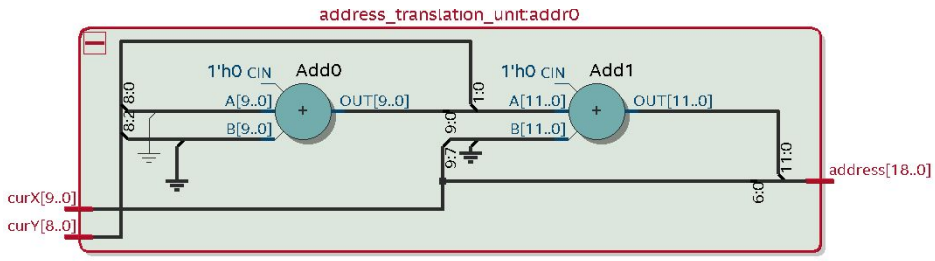
Module: generate_gauss_plane.sv
Inputs: clk, reset, start, jmp_julia, [63:0] dX, [63:0] dY, [63:0] X, [63:0] Y, [63:0] Magnification
Outputs: [63:0] re_c, [63:0] im_c, done
Description: Converts input coordinates of the form (X,Y), offset pairs of the form (dX, dY), and magnification coefficient into a point c on the complex plane with a real-and imaginary-component using a projection matrix and linear transformations.
Purpose: Implements logic needed to map pixels from VGA space onto the complex plane and apply shifts in perspective in response to panning over different regions of the fractal or different degrees of magnification. Used by other modules to create linear mapping between VGA space and complex-coordinate space.

Module: generate_fractal.sv
Inputs: clk, reset, [63:0] re, [63:0] im, ready, ack
Outputs: [9:0] escape, done
Description: Calculates escape value associated with the point described by values stored in re and im, respectively, by recursively evaluating mandelbrot polynomial and testing for convergence of result.
Purpose: Implements logic needed to generate fractal and test for convergence about a given point c on the complex plane whose components are given as input. Used by Fractal Core Interface module to associate escape values with pixels being drawn to the display.

Module: Magic_Numbers.sv
Inputs:
Outputs:
Description:
Purpose: Defines hard-coded constants and macros used by other modules during calculations.

Module: altpll_module.sv
Parameters: INPUT_PERIOD_NS, ATTENUATE_BY, GAIN_BY
Inputs: clk_in
Output: clk_out
Description: Creates generated clock signal clk_out by dividing the frequency of clk_in by a factor of ATTENUATE_BY and multiplying the frequency of clk_in by a factor of GAIN_BY.
Purpose: Used by CFPU and FPU modules to drive calculations at a higher frequency so timing constraints of VGA are met and calculations are performed at high speeds.

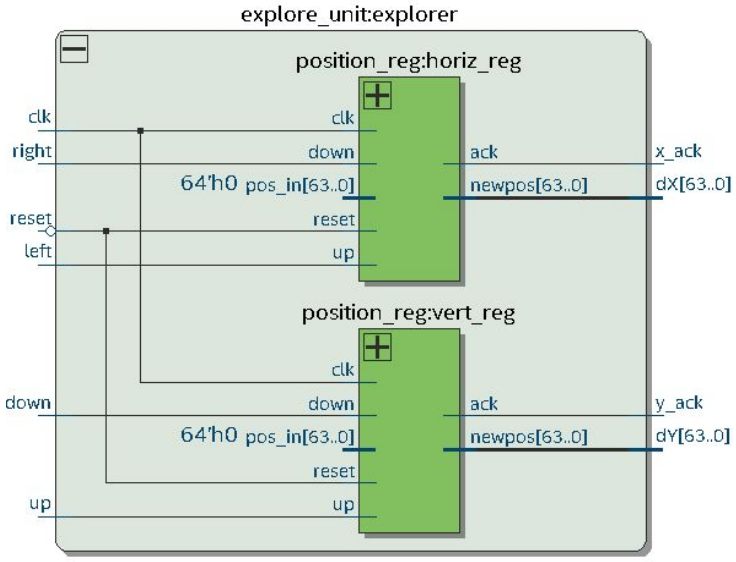
Module: address_translation_unit.sv
Inputs: [9:0] curX, [9:0] curY
Outputs: [18:0] address
Description: Converts pixel located at coordinates described by curX and curY on VGA display into a memory address in the RAM module used by the VGA frame-buffer component of design. Performs calculation that address = curY * 640 + curX.
Purpose: Implements logic to calculate address associated with a pixel at a given coordinate. Used by VGA frame-buffer to determine read and write addresses for memory operations.



RTL block diagram of address_translation_unit module

Module: pxlram.v
Inputs: [3:0] data, [18:0] rdaddress, [18:0] wraddress, rdclock, wrclcok, wren, rden
Outputs: [3:0] q
Description: Instantiates dual-port RAM module driven by two different read and write clocks to load data into memory pointed to by wraddress and retrieve data from memory pointed to by rdaddress.
Purpose: Used by VGA frame-buffer module to store escape values associated with pixels and derive color assignments when drawing a frame. Module was generated using a Quartus Megafunction wizard.

Module: explore_unit.sv
Inputs: clk, reset, up, down, left, right
Outputs: [63:0] dX, [63:0] dY, x_ack, y_ack
Description: Maintains internal registers for offset values in x- and y-direction from center of the screen. Sends acknowledgement signals when internal registers are updated.
Purpose: Used by top-level module to maintain current coordinates of center pixel when drawing fractal. Implements logic for increasing or decreasing offset values along the horizontal and vertical axis.



RTL Block Diagram of explore_unit module

Module: position_reg.sv
Parameters: SPEED = 100 // default
Inputs: clk, reset, [63:0] pos_in, up, down
Outputs: [63:0] newpos, ack

Description: Calculates new position given current position and a direction signal. Up signals increase the value of pos_in by SPEED. Down signals decrease the value of pos_in by speed. Generates acknowledgement signal once newpos is calculated.
Purpose: Used by explore_unit to implement registers for vertical and horizontal offsets, respectively.

Module: fractal_control_unit.sv
Inputs: clk, math_clk, reset, draw_frame, signewmag, up, down, left, right, magnify, demagnify, signewrow, signewframe, done_draw_esc, fractal_engine_done, [63:0] X_in, [63:0] Y_in, [9:0] SW
Outputs: sigvgadraw, [9:0] X_out,[9:0] Y_out, init_fractal_engine, draw_esc, fractal_engine_ack, sigack, go_up, go_down, go_left, go_right, go_magnify, go_demagnify
Description: Implements control logic for Fractal Core Interface component of the design. Coordinates actions and exchange of signals between various design components such as Fractal Engine component, VGA frame-buffer component, explore_unit module, and Magnification_Reg module.
Purpose: Implements control logic for the fractal interface and facilitates drawing of frames with calculations made by the fractal engine. Implements logical connections between modules on the datapath of the Fractal Engine and VGA frame-buffer with modules and signals instantiated in the top-level module of the entire design.

Module: adder_unit.sv
Inputs: clk, reset, start , ack, [63:0] re_a, [63:0] im_a, [63:0] re_b, [63:0] im_b
Outputs: done, [63:0] re_z, [63:0] im_z
Description: Implements complex addition arithmetic operation between two complex valued operands
Purpose: Used to calculate sums of complex numbers and represent the result as a complex number.

Module: multiplier_unit.sv
Inputs: clk, reset, start , ack, [63:0] re_a, [63:0] im_a, [63:0] re_b, [63:0] im_b
Outputs: done, [63:0] re_z, [63:0] im_z
Description: Implements complex multiplication arithmetic operation between two complex input operands using the identity that (a+bi)(c+di) = (ac-bd) + i(ad+bc).
Purpose: Implements complex-number multiplication algorithm logic and used to calculate products of complex numbers and represent the result as a complex number.

Module: magnitude_unit.sv
Inputs: clk, reset, start , ack, [63:0] re_in, [63:0] im_in
Outputs: done, [63:0] mag_out
Description: Implements complex squared magnitude arithmetic operation for a complex-valued input operand. Computes the squared sum of the square of each input component.
Purpose: Implements squared complex-number magnitude algorithm logic and used to calculate squared magnitudes of complex numbers and represent the result as a IEEE-754 scalar value when evaluating the mandelbrot polynomial about a complex point c.

Module: fractal_engine_control_unit.sv
Inputs: clk, reset, ack, ready, generate_coord_system_done, generate_fractal_done, [9:0] escape_in

Outputs: generate_coord_system, generate_fractal, done, [9:0] escape_out, get_coord_system, get_fractal, clear_fractal
Description: Coordinates generation of fractal_engine control signals based on state and input signals received.
Purpose: Implements state controller logic for Fractal Engine component of design. Used to interoperate the actions of generate_gauss_plane module and generate_fractal module with each other by exchanging acknowledgement signals between the two modules and generating control signals to transition these modules to the correct next-states.

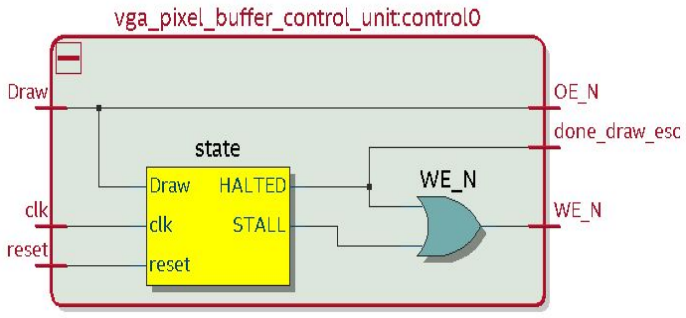
Module: fractal_engine.sv
Inputs: clk, reset, ack, [9:0] SW, [63:0] Magnification, [63:0] dX, [63:0] dY, [63:0] X, [63:0] Y
Outputs: [9:0] escape, reset_x, reset_y, done, [63:0] re_c_out, [63:0] im_c_out
Description: Instantiates modules tasked with generating points on the complex plane and assigning each point an escape value determined by the mandelbrot algorithm.
Purpose: Implements logic needed to regenerate escape values for each point within a fixed-size region each time the position or magnification are updated or reset signal is toggled. Used by Fractal Interface Core component of design to generate fractal about a given center coordinate and magnification coefficient.

Module: Mux.sv
Parameters: DATA_SIZE=8
Inputs: [DATA_SIZE-1:0] A, [DATA_SIZE-1:0]B, SELECT
Outputs: [DATA_SIZE-1:0] OUT
Description: Implements two-to-one multiplexer with input operands of width DATA_SIZE.
Purpose: Used to connect an input data line to the output port depending on value of select signal.

Module: vga_pixel_buffer.sv
Inputs: clk, reset, math_clk, color_clk, Clear, [63:0] curX, [63:0] curY, [15:0] escape_value, [9:0] drawxsig, [9:0] drawysig, vga_clk, Draw, draw_frame, vga_ready
Outputs: done_draw_esc, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, OE_N, WE_N
Description: Used to load escape values for a given pixel described by coordinates (curX, curY) into memory and assign color values to the pixel being drawn to VGA coordinates (drawxsig, drawysig). Manages memory read and write operations and determines which escape values are used to color a pixel depending on the status of VGA controller and Fractal Engine component of design. Determines read and write addresses for data in memory.
Purpose: Implements datapath for VGA frame-buffer and modules for translating output of Fractal Engine design component to RGB values used to color the pixel being drawn by the current VGA frame at that instance in time.

Module: vga_pixel_buffer_control_unit.sv
Inputs: clk, reset, Draw
Outputs: done_draw_esc, OE_N, WE_N
Description: Generates control signals used to perform memory operations and signals to the Fractal Interface Core component that pixel has been drawn to screen.

Purpose: Implements state controller logic for VGA frame-buffer components of design and sets OE_N and WE_N signals used to perform memory operations depending on state of Fractal Engine and Fractal Core Interface components of design.



RTL block diagram of vga_pixel_buffer_control_unit module

Module: colorize_pixel_unit.sv
Inputs: clk, vga_clk, reset, [7:0]escape_value, start, [9:0] drawxsig, [9:0] drawysig
Outputs: [7:0] R, [7:0] G, [7:0] B
Description: Assigns RGB color channel values to R, G, and B output signals based on escape value connected as input. Looks up RGB value to use when encoding color channels using a 16 entry color palette.
Purpose: Implements logic for assigning color values to RGB color channels based on escape value for the current pixel being drawn.

Module: Reg_unit.sv
Parameter: DATA_SIZE = 8
Inputs: CLK, reset, LD, [DATA_SIZE-1:0] Din
Outputs: [DATA_SIZE-1:0] Dout
Description: Implements N-bit register with synchronous reset.
Purpose: Used for storing data about escape values in VGA frame-buffer modules after it becomes available. Helps prevent invalid data from being used to encode RGB color channels.

Module: generate_gauss_plane_control_unit.sv
Inputs: clk, reset, start, convert_x_done, convert_y_done, magnify_x_done, magnify_y_done, update_x_scaling_done, update_y_scaling_done
Outputs: get_x_as_ieee, get_y_as_ieee, start_x_conversion, start_y_conversion, start_magnify_x, start_magnify_y, start_update_x_scaling, start_update_y_scaling, get_x_magnification, get_y_magnification, get_update_x_scaling, get_update_y_scaling, done
Description: Generates control signals needed to perform operation of translating VGA coordinates to complex-plane coordinates. Facilitates transitions between states based on input signals by generating the corresponding output signals.
Purpose: Implements state controller logic for generate_gauss_plane module used when converting VGA space coordinates to complex-plane coordinates. Generates control signals and acknowledgement signals used to communicate internal state with other modules and design components.
Module: fractal_generator_control_unit.sv

Inputs: clk, reset, ready, ack, done_sq, done_sum, done_mod, [9:0] escape_in, [63:0] re_in, [63:0] im_in, [63:0] mod_in, [63:0] re_in_wire, [63:0] im_in_wire, [63:0] mod_in_wire,
Outputs: init_sq, init_sum, init_mod, get_mod, ack_sum, ack_sq, done, [9:0] escape_out, [63:0] re_out, [63:0] im_out, [63:0] mod_out
Description: Generates control signals needed to perform iterations of mandelbrot algorithm and compute escape values associated with a point c on the complex plane. Facilitates transitions between states by generating corresponding output signals and connecting output wires to input wires based on current state.
Purpose: Implements state controller for fractal_generator module used to assign escape values to each point c on the complex-plane within the viewing region.Generates control signals and acknowledgement signals used to communicate internal state with other modules and design components.

Module: fractal_core_interface.sv
Inputs: clk, math_clk, vga_clk, color_clk, reset, Clear, up, down, left, right, magnify, demagnify, signewmag, draw_frame, [1:0] KEY, [63:0] magnification, [63:0] dX_in, [63:0] dY_in, [9:0] drawxsig, [9:0] drawysig, signewcurX_in, signewcurY_in, [63:0] curX_in, [63:0] curY_in
Outputs: go_up, go_down, go_left, go_right, go_magnify, go_demagnify, sigack, [63:0] curX_out, [63:0] curY_out, [7:0]VGA_R, [7:0] VGA_G, [7:0] VGA_B, OE_N, WE_N
Description: Instantiates design components used to display colorful images to the VGA display based on escape value corresponding to current pixel being drawn. Instantiates design components used to map VGA display pixels to points on complex-plane. Facilitates regeneration of fractal image each time the magnification changes or position changes. Creates connections between signals used by design components with signals and modules used in the top-level module of the entire project.
Purpose: Implements connections between design components and top-level design entity. Manages the transition between states and actions of the Fractal Engine component and VGA frame-buffer component. Generates signals used to modify current position of center coordinate and magnification-coefficients.

Module: double_adder.sv
Inputs: clk, reset, [63:0] a_in, [63:0] b_in, a_in_done, b_in_done, z_out_ack
Outputs: a_in_ack, b_in_ack, z_out_done, [63:0] z_out
Description: Calculates the sum of two floating-point input operands and outputs result as a 64-bit floating point value.
Purpose: Used by CFPU modules and Fractal Engine unit to compute sums of operands represented in IEEE-754 floating-point representation.

Module: Vga_Controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0]DrawY
Description: Generates VGA signals used for the timing and drawing of a frame. Input Clk is main FPGA clock and derives signals for hs, vs, pixel_clk, blank and sync signals. These outputs control the deflection of the drawing beam and the transitions from the last pixel of a frame to the first pixel or the last pixel of a row to the first pixel in the next row.
Purpose: This module encapsulates the logic needed to operate the VGA interface and display images to the external display during each frame interval.

Module: double_multiplier.sv
Inputs: clk, reset, [63:0] a_in, [63:0] b_in, a_in_done, b_in_done, z_out_ack
Outputs: a_in_ack, b_in_ack, z_out_done, [63:0] z_out
Description: Calculates the product of two floating-point input operands and outputs result as a 64-bit floating-point value.
Purpose: Used by CFPU modules, Fractal Engine unit, and Magnifier_Reg module to compute products of operands represented in IEEE-754 floating-point representation.

Module: int_to_ieee.sv
Inputs: clk, reset, [63:0] a_in, a_in_done, z_out_ack
Outputs: a_in_ack, z_out_done, [63:0] z_out
Description: Calculates the 64-bit IEEE-754 floating-point representation of the input operand taken to be an integer and outputs the result as a 64-bit floating point value.
Purpose: Implements logic for converting between integer representation and 64-bit floating-point representation. Used by Fractal Engine modules to convert x- and y-displacement terms into floating-point representation so that the magnification-coefficient may be applied to calculate scaled output values when generating points on the complex plane.

Module: ieee_to_int.sv
Inputs: clk, reset, [63:0] a_in, a_in_done, z_out_ack
Outputs: a_in_ack, z_out_done, [63:0] z_out
Description: Calculates the integer representation of a 64-bit IEEE-754 floating-point input operand and outputs result as a 64-bit integer value.
Purpose: Implements logic for converting 64-bit double-precision floating-point values into integer representation.

Module: lab8_soc.v
Inputs: [31:0] curx_w0_in_port, [31:0] curx_w1_in_port, [31:0] cury_w0_in_port, [31:0] cury_w1_in_port, [31:0] mag_w0_in_port, [31:0] mag_w1_in_port, clk_clk, reset_reset_n, signewcurx_in_port, signewcury_in_port, signewmag_in_port, spi0_MISO, usb_gpx_export, usb_irq, export, [1:0] key_external_connection_export
Outputs: [31:0] curx_w0_out_port, [31:0] curx_w1_out_port, [31:0] cury_w0_out_port, [31:0] curx_w1_out_port [31:0] mag_w0_out_port, [31:0] mag_w1_out_port, signewcurx_out_port, signewcury_out_port, signewmag_out_port, [7:0] keycode_export, sdram_clk_clk, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SS_n, usb_rst_export, [13:0] leds_export, [15:0] hex_digits_export, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n
Inout: [15:0] sdram_wire_dq
Description: Defines the interface for the NIOS-II SoC. Data exchanges between the project hardware component and software components occur between these connections to the SoC.
Purpose: Implements connections between hardware and software components of the design. Implements logic for retrieving keycodes generated by an external USB keyboard from the top-level module of the

project. Also implements connections for viewing the current x-coordinate, y-coordinate, and magnification coefficient from the NIOS-II console. Additional connections are present to implement the feature of being able to set the x- or y-coordinates and magnification-coefficient from software but these features were never fully realized in the design.

VII. Design Resources and Statistics

LUT	25,447
DSP	630
Memory (BRAM)	1,437,696
Flip-flop	15,826
Frequency	75.77 MHz
Static Power	105.70 mW
Dynamic Power	650.66 mW
Total Power	777.92 mW

Design Analysis statistics and measurements.

VIII. Conclusion

This final project proved to be challenging because of the different timing constraints and resource constraints presented by the different aspects of the design. Often, I found myself trading-off between critical path length and resource consumption for a given module. For example, one bottleneck I struggled to overcome was the limitation to the size of the color-palette. Because each pixel has its own memory address, the number of bits in memory per pixel was reduced to 4 in order to fit the design onto the FPGA. At first, I thought the timing signals generated by the VGA interface were incorrect when the output image appeared distorted, but after further debugging I determined that the RAM module being used could not accommodate each unique pixel and that memory constraints rather than timing constraints were the source of the error. In turn, I had to reduce the data width of entries stored in the RAM in order to fit each pixel address into memory. Consequently, the displayed image can only be generated using a palette with 16 different RGB values. As an attempt to overcome this limitation, I thought I would calculate a cosine wave in hardware[7] and evaluate the cosine using the escape value as an argument and then scale this result to be within the range [0, 255] for each RGB color channel. Then, I would store the top 4 bits which are actually used by the FPGA's DAC to color the pixel into memory. The additional overhead from these extra calculation steps ended up making the design so large it would not fit on the FPGA. Even when I got the design to fit by decreasing the level of precision, the coloring component failed to meet the VGA timing constraints because the critical path of the color signals became too long. Another approach I attempted to overcome this limitation was to draw the output image using interlacing techniques. To implement this, I tried to introduce a new frame clock which operates at a higher frequency such that scanlines with even draw-y indices were loaded into memory and drawn during a first pass over the display and scanlines with odd draw-y indices were loaded into memory and drawn during a second pass. This way, only half of the pixel escape values being drawn would need to be in memory at a given time, ultimately reducing the number of memory addresses in half and allowing more data to be stored at each address. However, with the deadline approaching, I was hesitant to iron out the bugs in this implementation and reverted back to using a 4-bit width memory. Other bottlenecks I encountered occurred when the magnification-coefficient became significantly small. The amount of time needed to

[7] https://kierdavis.com/cordic.html

perform calculations increased dramatically and the VGA frame rendering performance was poor. To overcome this, I attempted to approximate the squared magnitude of the point c on the complex plane used in determining escape values by computing the logarithm of the squared modulus and evaluating its order of magnitude. Again, this choice increased the critical path too dramatically to be used in the final implementation. To improve the overall rendering speed of the output image to the VGA, I ran the CFPU and FPU core modules at higher clock frequencies than the main clock and VGA clock. In particular, I found that driving the calculation modules at 150 MHz was the maximum frequency at which the design would operate before becoming unable to generate and draw pixel colorings during a frame interval.

Another challenging feature of the design was implementing the state machines for the Fractal Engine and its subcomponents. Deriving the mapping function from the VGA coordinate system to the complex-coordinate system was a challenge in-itself because my debugging efforts and ability to render an image on the screen depended on the correctness of the perspective transformation functions used to translate between the two coordinate systems. Reading into how frameworks such as OpenGL perform similar projections helped me gain a better understanding of how to perform my own linear transformation. Additionally, implementing the mandelbrot algorithm was challenging because its state controller generates signals to reuse the current output value as the next input in order to evaluate the mandelbrot polynomial recursively. This design aspect was challenging to implement in hardware because it introduced a feedback loop to the module design which could corrupt input and output signals when state transitions were not carefully taken and output values were not evaluated in the correct order. While implementing recursion in software, one has more freedom to define the mechanisms for detecting base cases and the actions taken as a result. However, when implementing a recursion in hardware, the base cases need to be detected by the state controller and the actions taken become a part of the state transition sequence within an execution cycle. Although difficult, both of these challenges posed interesting problems to solve and my design choices to address these challenges helped shape the overall design and performance of the project.

The lab documentation and materials provided for the final project were sufficient for implementing the VGA frame-buffer of the design and configuring the SoC component of the design to interface with signals generated by the FPGA hardware. Because of the nature of the final project, I feel the expectations and timeframe were appropriate. The experiences gained from prior labs definitely established a strong enough foundation of digital system design knowledge and familiarity with SystemVerilog and tools such as Quartus and the Platform Designer to the point where taking on the task of designing and implementing a final project in a relatively short timespan did not feel extremely overwhelming or that I was unprepared due to a lack of experience and knowledge with these FPGA application design and synthesis tools.