FPGA Graphics Engine: Rendering a Rotating Tetrahedron

Ben Sheppard, Andrew Belles

August 29, 2025

Contents

Abstract

The field of computer graphics is rich in opportunity to develop skills with computational programming as well as control logic. For our project we sought to build our own graphics engine from the ground up. The graphics engine we have built renders a tetrahedron that is able to be rotated on the x, y, and z axis as well as rotated on joint axes simultaneously (x and y, etc.) via keyboard inputs. A UART receiver handles inputs of keys WASDQE for single axis rotations and IJKLUO for double-axis rotations. A custom graphics linear algebra module handles the rotation and projection of points from some local space (3D) down to a viewport/screenspace (2D); which is communicated to a graphics manager that then algorithmically determines which pixels make up the connecting lines between points, drawing the shape to the screen. These modules are controlled by a top level shell and controller that handles the timing and communication between modules. Future improvements we wish to make include more efficient utilization of FPGA resources to allow for more arbitrary, complex shapes and being able to color/shade the faces of an arbitrary shape.

1 Systems Overview

Our goal for the engine was to be able to render a projection with depth of a 3D tetrahedron with the user being able to rotate the shape in real time. The FPGA should receive inputs from a keyboard using UART and request updates from the math manager on the tetrahedron. The math manager will provide these updated by rotating continuously and smoothly on the requested axis. For each applied rotation the projection onto our viewport will be computed and each coordinate pair (x_i, y_i) will be packed into a 16 bit logic vector (x :15 downto 8, y : 7 downto 0) and sent to be drawn upon request by our graphics manager. The graphics manager will take the point packets generated by the math manager and utilize Bresenham's line algorithm to discretely locate each pixel that lines on a line between two points. The manager sends the data serially as each pixel is computed to a module with dual BRAMs. It writes the illuminated pixels to the buffer that is not being displayed, and then flips which buffer is being displayed. The system should appear visually smooth and respond quickly and with low latency to user inputs. The actionable inputs are as follows,

- W: +Z rotation
- A: +Y rotation
- S: -Z rotation
- D: -Y rotation
- Q: +X rotation
- E: -X rotation
- I: +X and +Y rotations
- J: +Y and +Z rotations
- K: -X and -Y rotations
- L: -Y and -Z rotations
- U: +X and +Z rotations
- O: -X and -Z rotations
- R: Resets the render to its initial state

Our design specifically utilizes the aforementioned Math and Graphics Managers which act as their own shells for the respective computation and control logic that they "oversee". Since we want the system to decouple the math from drawing logic we opted for the Graphics Manager to see the Math Manager as a black box where the Top Level Shell can request an update to points from the Math Manager and upon completion a new packet is sent to the Graphics Manager to be drawn.

1.1 Top-Level Block Diagram

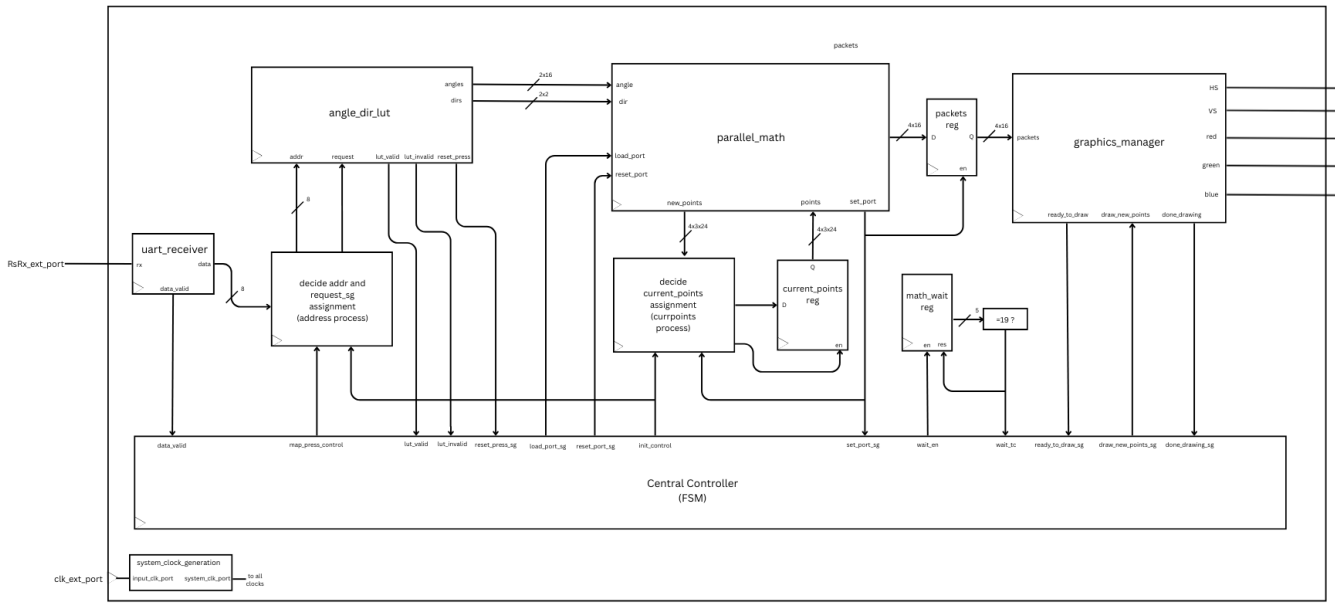

Figure 1: Top-level block diagram of the Central Controller datapath.

1.2 Description of Ports

NB: We define all `array_t` datatypes in **Appendix A**.

- clk_ext_port: (in : std_logic)
  - This is the 100 MHz input from the FPGA internal clock. We divide this down to 25 MHz using the system_clock_generation module, which provides the clock for all other components.
- RsRx_ext_port: (in : std_logic)
  - This is the serial input from the Basys3 USB-UART serial bridge. We use the PuTTy application to relay a keyboard button press to the FPGA using the UART protocol. The received message is interpreted by the uart_receiver module.
- HS: (out : std_logic)
  - Horizontal sync signal for the 640x480 VGA monitor. This is sent to the HS pin on the VGA connector through the on-board port.
- VS: (out : std_logic)
  - Vertical sync signal for the monitor. This is sent to the VS pin on the VGA connector through the on-board port.
- red/green/blue: (out : std_logic_vector(3 downto 0))
  - 4-bits for each of red, green, and blue. They are sent to their respective pins on the VGA connector through the on-board port.

1.3 Description of Components

- system_clock_generation:
  - Divides the 100 MHz FPGA clock to 25 MHz using a divider ratio of 4. Does this by toggling the output every terminal count (divider ratio / 2) . Puts system_clk on the clocking network using BUFG.
- uart_receiver:
  - Receives the UART ASCII code from a key press. Uses a 9600 baud rate to read a start bit (0), 8 data bits, and a stop bit (1) , with idle bits (1) in between each code.
- angle_dir_lut:
  - Maps the ASCII code from the keyboard to an angle and direction to rotate, which is sent to the parallel math module (math manager). For the special case when the reset key (R) is pressed, tells the central controller to re-initialize, resetting the tetrahedron.
- parallel_math
  - Contains the matrix multiplication to rotate the four vertices for a given angle and direction. Also projects the 3D points into 2D space using perspective and sends those points to the graphics manager for drawing.
- graphics_manager
  - Receives 4 points from the math manager and draws single-pixel lines for each combination of them to render a tetrahedron on the monitor.
- address process
  - Decides whether to send angle_dir_lut a special code for initialization or the ASCII code from the uart_receiver. It also asserts a signal to request an angle/direction from the LUT.
- currpoints process
  - Decides whether to send the math manager hard-coded initialization points or points stored by the current_points register for update.
- math_wait register
  - Tells the controller when 20 clock cycles has passed so that it can start the math phase.
- packets register
  - Ensures that the packet (4 points) sent from the math manager to the graphics_manager for drawing is stable.

2 Technical Description

2.1 VGA_Controller (vga_controller.vhd)

2.1.1 Description of Ports

- clk (in, std_logic):
  - 25 MHz system clock
- video_on (out, std_logic):
  - Only high when pixel_x and pixel_y are within the region of the screen where the video is on. For pixel_x, this is between 0 and 639, and for pixel_y this is between 0 and 479.
- h_sync (out, std_logic):
  - Active low when pixel_x is between 656 and 751, which tells the monitor that we have reached the end of the line and must go to the next line.
- v_sync (out, std_logic):
  - Active low when pixel_y is between 490 and 492, which tells the monitor that we have reached the bottom of the monitor and must go back to the top.
- pixel_x (out, std_logic_vector(9 downto 0)):
  - 8 bit binary representation of the current x location of the monitor's horizontal scan. Ranges from 0 to 799. Increments with each edge of the 25 MHz VGA clock until it reaches 799, when it is reset to 0.
- pixel_y (out, std_logic_vector(9 downto 0)):
  - 8 bit binary representation of the current y location of the monitor's vertical scan. Ranges from 0 to 520. Increments when pixel_x reaches 799 (its reset). Resets to 0 when it reaches 520.

2.1.2 RTL Diagram

The hscan and vscan counter registers are initialized to all zeros. Because the system clock frequency (25 MHz) has the same frequency as the VGA clock, the hscan counter simply increments with each rising system clock edge and resets at 799. The vscan counter increments with every hscan reset, representing moving to the next line. The VGA controller asserts video_on if these values are between 0 and 639, and 0 and 479, respectively. Pixel_x and pixel_y are the raw hscan and vscan values at any given time. Finally, h_sync and v_sync are asserted (active low) between hscan and vscan values of 656 and 751, and 490 and 492, respectively. These blocks (on the right side of the diagram) are simple combinational processes.

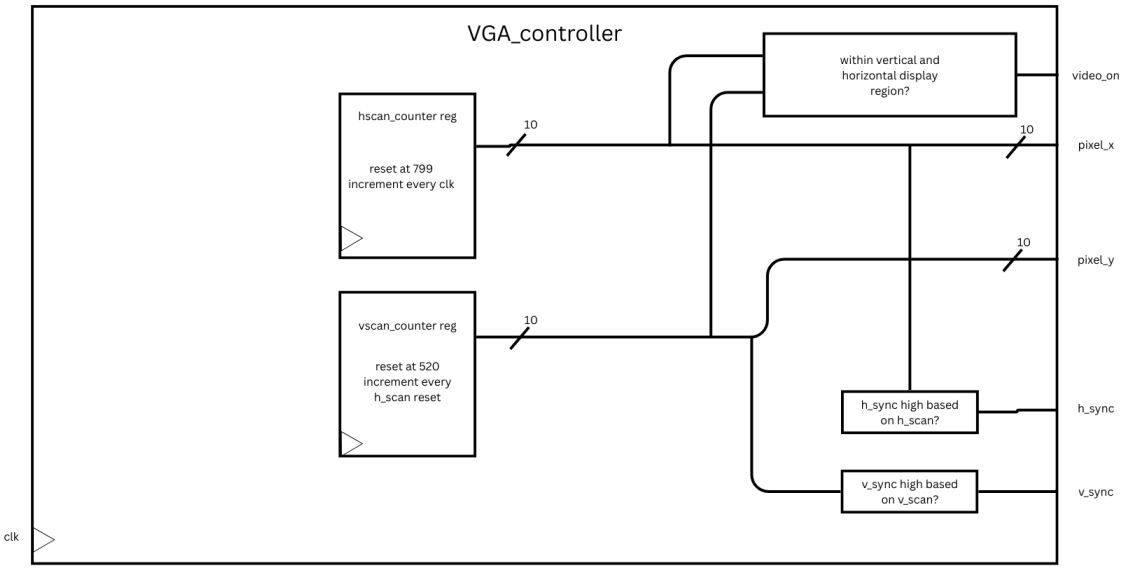

Figure 2: VGA Controller RTL

2.2 Bresenham (bresenham.vhd)

We chose to use the Bresenham algorithm because it connects a straight-looking line between two pixels by only requiring simple additions and bit shifts rather than floating-point arithmetic. The algorithm iteratively steps, deciding whether to step in x, y, or both. First $dx = x_1 - x_0$ and $dy = y_1 - y_0$ are computed, which are the horizontal and vertical extents of the optialline. By applying a sign correction, any slope can be processed. Then, $err = dx - dy$ is computed, which provides a notion of error between the actual line and the current pixel's position. It then computes $e2 = 2 * err$, critical for avoiding fractional error (err is always either an integer or a half-integer such as 1.5, so multiplying by 2 makes it an integer). $e2$ is then compared to dx and dy. If $e2 > dy$, we have crossed a threshold, so need to step in the x direction. If $e2 < dx$, then we additionally need to step in the y direction. By iterating until x and y are equal to x_1 and y_1, the algorithm creates a straight line between (x0, y0) and (x1, y1), asserting done when this condition is met.

2.2.1 Description of Ports

- clk (in, std_logic):
  - 25 MHz system clock
- start (in, std_logic):
  - Starts the algorithm. Endpoints must be valid when start is asserted.
- reset (in, std_logic):
  - Resets the algorithm. Not used in this project, but was part of original Bresenham module design.
- x0, y0 (in, std_logic_vector(7 downto 0):
  - 8 bit binary representation of x and y coordinates for first endpoint. Range from 0 to 255, with (0, 0) in top left and (255, 255) in bottom right.
- x1, y0 (in, std_logic_vector(7 downto 0):
  - 8 bit binary representation of x and y coordinates for second endpoint. Range from 0 to 255, with (0, 0) in top left and (255, 255) in bottom right.
  - x, y (out, std_logic_vector(7 downto 0):
- done (out, std_logic):
  - Control signal stating that all pixels have been drawn and we are ready for the next start
- plot (out, std_logic):
  - Signal high while the algorithm is actively providing pixels on x, y output that should be illuminated

2.2.2 RTL Diagram

Note that the three RTL blocks are screenshots directly from Edwards. The RTL of this algorithm was unlike any other we have seen in this class, using variables within a process. Block 1 computes dx and dy and checks if the sign flip discussed above should be applied. Block 2 computes err from dx and dy and then applies a left shift to compute $e2$. Block 3 stores the current point (x and y) in two registers and checks whether we should increment or decrement x and/or y. It also checks whether (x, y) are equal to (x1, y1), which results in done being asserted.

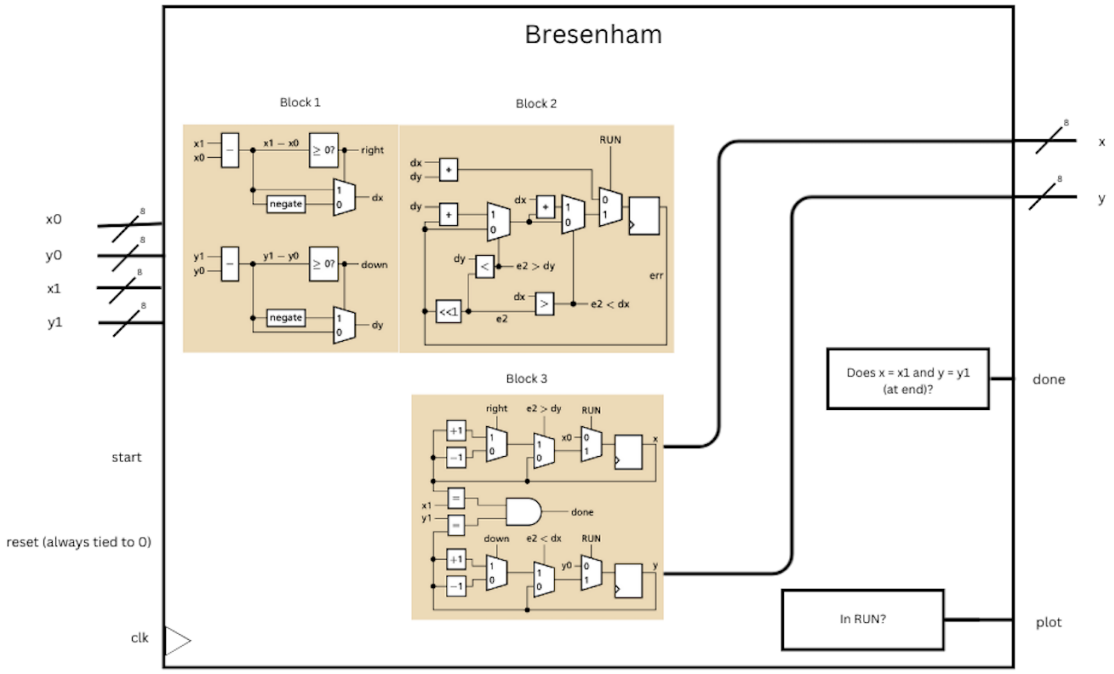

Figure 3: Bresenham Datapath

2.2.3 FSM Diagram

The FSM of this algorithm was unlike any other we have seen in this class, with a structure that closely intertwines the datapath and FSM. The IDLE state waits until start is asserted, when Block 1 is executed. It then transitions to the RUN state, where blocks 2 and 3 are repeatedly executed until done is asserted, when it transitions back to IDLE.

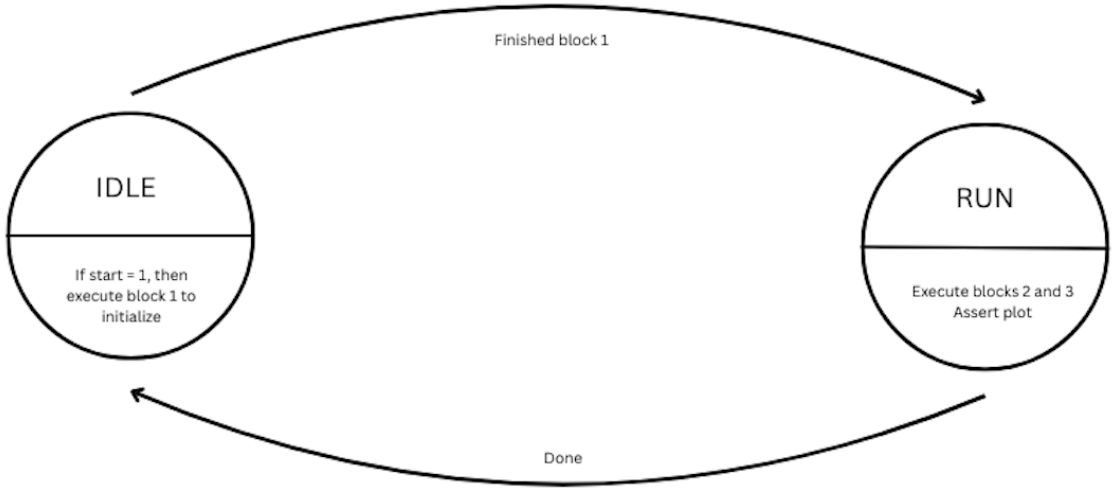

Figure 4: Bresenham FSM

2.3 Bresenham_Receiver (bresenham_receiver.vhd)

At a high level, the Bresenham_Receiver has three main functions. First it directs the encapsulated Bresenham sub-module to execute its algorithm 6 times, one for each combination of the 4 vertices of the tetrahedron. Each

correct illuminated pixel is sent on the rising edge of a clock directly from the Bresenham sub-module. Second, it communicates with the framebuffer module telling it to clear its back buffer (the one that is not currently being displayed) before starting the Bresenham algorithm activates. This ensures that each draw is done onto a clean buffer of zeros. Finally, it notifies the central controller when it is finished drawing so that it can go to its next state.

2.3.1 Description of Ports

- clk (in, std_logic):
  - 25 MHz system clock
- vertices (in, array_4x16_t):
  - Packet of 4 points to draw from the math manager. Each of the 4 row of the matrix is a separate point. The 8 MSBs of the row specify the x coordinate and the 8 LSBs of the row specify the y coordinate. The coordinate system has (0, 0) at the top left and (255, 255 at the bottom right).
- new_vertices (in, std_logic):
  - Tells the Bresenham receiver to draw a new tetrahedron defined by the points given in packets
- clear_fulfilled (in std_logic):
  - Tells the Bresenham receiver that the back buffer of framebuffer has been cleared and it can proceed to start sending points that it can write to the back buffer
- x, y (out, std_logic_vector(7 downto 0)):
  - Represents a singlepixel in the (0, 0) to (255, 255) coordinate system. Sent to the framebuffer where it is written as a 1 in the back buffer.
- clear_request (out, std_logic):
  - Sends a request to the framebuffer to clear the back buffer before it starts computing pixels to illuminate.
- tet_drawn (out, std_logic):
  - Tells the framebuffer that it is done sending pixels so that it can prepare to swap the front and back buffer
- load_mem (out, std_logic):
  - Tells the framebuffer that it is actively sending pixels that it should be writing to its back buffer.

2.3.2 RTL Diagram

The x_points and y_points registers separately hold the four x and y points in our array_4x8_t datatype for easy access by the Bresenham sub-module. The counter register counts from 0 to 5, telling the receiver FSM when it has computed all 6 combinations of points. These combinations, i.e. point 1 to point 2, point 1 to point 3, etc., are hard-coded and sent to the Bresenham sub-module using a case statement conditioned on this counter. The new_vertices, clear_fulfilled, clear_request, tet_drawn, and load_mem control signals are managed by the FSM.

2.3.3 FSM Diagram

The FSM begins in IDLE, where it continuously resets the counter and loads new points into the x and y point registers. When new_vertices is asserted, it leaves IDLE, deasserting load_new_points, which stores the points present on the vertices input at that time. In the Clear Buffer (CB) state, the module requests for the framebuffer's back buffer to be cleared, only transitioning to ACTIVATE when it receives a clear_fulfilled signal. For the first run of the algorithm, counter will be 0, so x_0, x_1, y_0, y_1 will be indexed from the x and y points register. Start_bres is asserted, which begins the sub-module's algorithm. x and y points are sent in the COMPUTE state until done_bres. If the counter is not up to 5, it will be incremented and the process will be repeated with two different points. If the counter is at 5, it will be reset and the FSM will transition to DONE, asserting tet_drawn before transitioning back to IDLE.

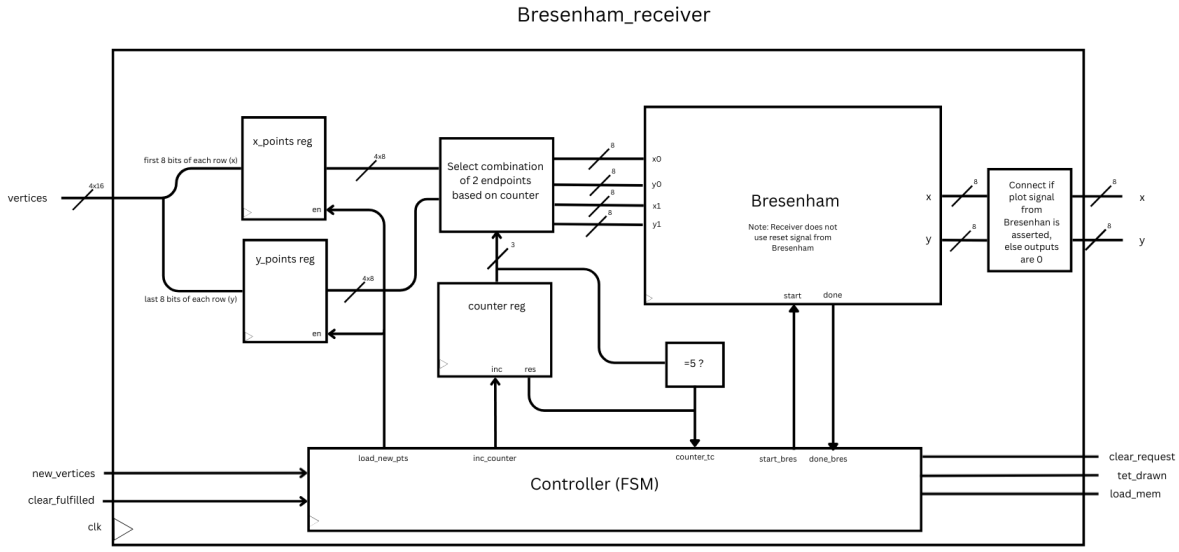

Figure 5: Bresenham Receiver Datapath

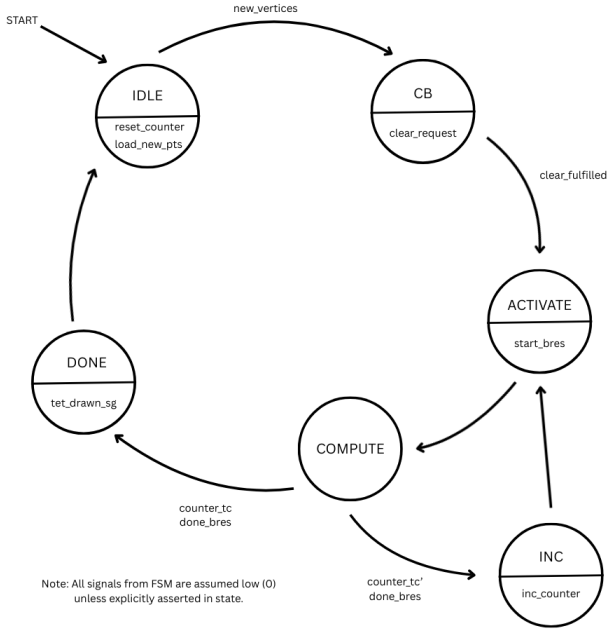

Figure 6: Bresenham Receiver FSM

2.4 Framebuffer (framebuffer_v2.vhd)

The framebuffer contains two BRAMs, a front buffer, which is read to determine whether each pixel on the monitor is illuminated, and a back buffer, which can be written to without disrupting the front buffer. After writing to the back buffer, the two buffers are swapped during a blanking region. Additionally, the module contains logic to determine the correct read/write addresses for the front/back buffers, clear the back buffer (write zeros to all memory locations) before writing a new tetrahedron to it, and assert the write enable on the back buffer. Finally, the module communicates with the central_controller and Bresenham_receiver to activate at the write time and write to the correct memory addresses.

2.4.1 Description of Ports

- HS_in, VS_in (in, std_logic):
  - Horizontal and vertical sync from vga_controller. Sent into graphics_manager because they need to be pipelined due to BRAM read delay, and it is clearer to pipeline all in one place.
- video_on (in, std_logic):
  - Video_on from vga_controller. Sent into graphics_manager because needs to be pipelined due to BRAM read delay, and it is clearer to pipeline all in one place.
- pixel_x, pixel_y (in, std_logic_vector(9 downto 0):
  - Pixel_x and pixel_y from the vga_controller are used to compute the read address from the front buffer (currently displayed on screen).
- write_x, write_y (in, std_logic_vector(7 downto 0):
  - Write_x and write_y from the Bresenham_receiver are used to compute the write address for the back buffer when writting the tetrahedron.
- write_en (in, std_logic):
  - Tied to the back buffer when writing to it. Asserted when the write_x and write_y represent valid pixel locations that draw a tetrahedron
- tet_drawn (in, std_logic):
  - Asserts when the tetrahedron is fully drawn, so it can prepare to swap the two buffers and display the new tetrahedron.
- clear_request (in, std_logic):
  - Asserts when the Bresenham_receiver has new vertices so requests that the back buffer is cleared
- VGA_HS, VGA_VS (out, std_logic):
  - Horizontal and veritcal sync outputs directly to the VGA connector on the board. 4 cycles behind HS_in and VS_in
- VGA_out (out, std_logic_vector(11 downto 0)):
  - 12-bit RGB value containing data to be sliced up by graphics_manager and drawn to red, green, and blue
- clear_fulfilled (out, std_logic):
  - Tells Bresenham_receiver that the framebuffer is done clearing the back buffer, so it is ready to receive pixels to write.
- ready_to_draw (out, std_logic):
  - Tells central_controller that the framebuffer is in IDLE, so is ready to draw
- done_drawing (out, std_logic):
  - Tells central_controller that the framebuffer is done drawing, so it should move on to its next state

2.4.2 RTL Diagram

At the core of the framebuffer module are the two BRAM blocks. Each are accessed with a unique address, buff0_addr and buff1_addr. These addresses are determined with the help of three processes: raddr, waddr, and addr_logic. The raddr process converts pixel_x and pixel_y, a 2D position which is being written to on the monitor, to a 1D address ranging from 0 to 65536. Furthermore, it only assigns read_addr when x and y are within the 256x256 window in the center of the monitor. The equation we used for this translation is below:

$$read\_addr = (y - 112) * 256 + (x - 192) \tag{1}$$

Similarly, the waddr process converts write_x and write_y to a 1D address; however, this equation is even simpler because write_x and write_y are already guaranteed to range from 0 to 256:

$$write\_addr = y * 256 + x \tag{2}$$

The addr_logic process wires up buff0_addr and buff1_addr. It wil always assign the read_addr to the front buffer. If the FSM is in the clearing state, it will assign the back buffer's write address to clear_addr, which counts from 0 to 65536. If it is not in a clearing state, it will assign the back buffer's write address to write_addr. The front buffer is stored in the front_buffer register, which can be toggled by the controller.

The write_data port of both BRAMs is connected to the inverted value of the clearing control signal since during the clearing period, we want to write 0, but during the drawing period we want to write 1. We assert the wea on the back buffer if we are clearing OR [if write_en is asserted AND we are receiving]. Finally, the controller receives the blanking control signal whenever VS_delayed is deasserted, since that means that we are outside of the active monitor region.

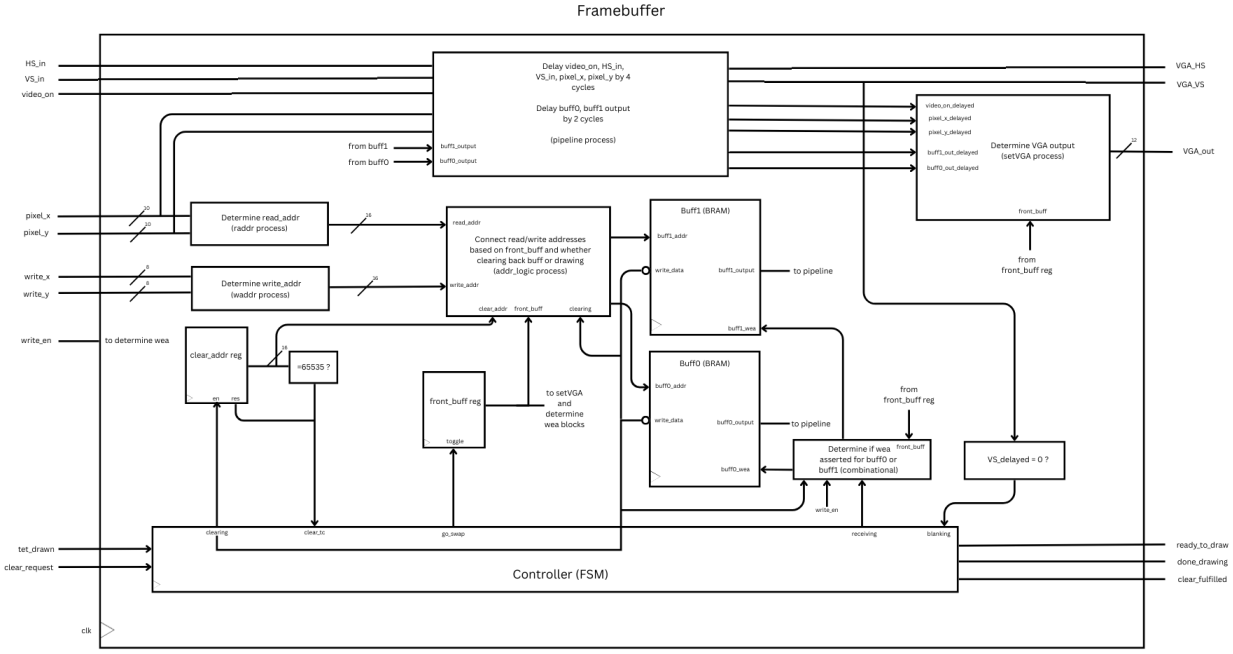

Figure 7: Framebuffer Datapath

2.4.3 FSM Diagram

The FSM begins in the IDLE state, outputting to the central_controller that it is ready to draw. If the Bresenham_receiver requests that the back buffer is cleared (it does this after receiving a new packet from the math manager), the FSM transitions to the Clear Back (CB) state and begins clearing, writing a 0 to every memory address. When it finishes clearing, it signals to the Bresenham_receiver that the clear was fulfilled in the CLEARED state, and then enters the RECEIVE state, where it writes the tetrahedron to the back buffer. When fully drawn it,

waits in WAITING for a blanking region and then swaps which buffer is in front so that in the next active region, the monitor will display the new shape. After swapping it tells the central_controller that it is done in the DONE state before returning to IDLE.

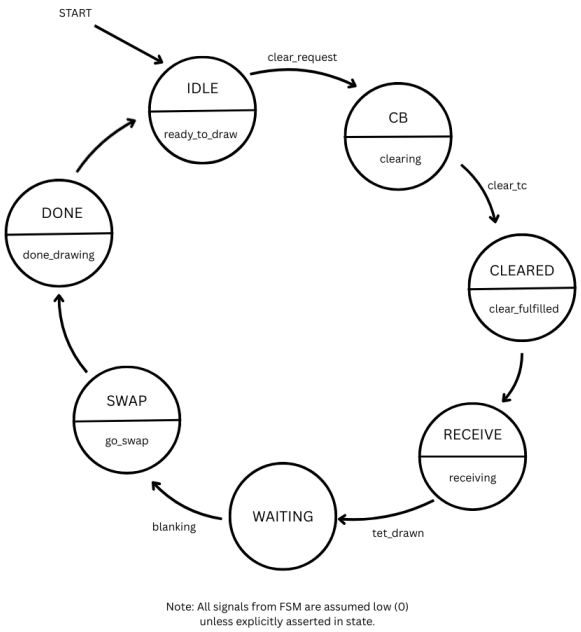

Figure 8: Framebuffer FSM

2.4.4 Description of Memory

Each BRAM block is instantiated using Vivado's Block RAM IP Core. They are single-port with width of 1 and depth of 65536, which provides enough locations for a 256x256 framebuffer. At a given location, the memory is a 0 if the pixel is off and a 1 if the pixel is on (along the tetrahedron edges). Because it is single-port RAM, it cannot be read and written at the same time, which is why we used a dual buffer setup. The RAM has a 2 cycle read latency, which means that the output changes 2 cycles after the addr changes. Because of this, we had to incorporate pipelining into our design so that the read values would be drawn to the correct pixel on the monitor.

2.5 Graphics Manager (graphics_manager.vhd)

At a high level, the graphics manager is responsible for receiving a packet of four points from the math manager, performing the Bresenham line-drawing algorithm between each combination of points, and drawing the resulting tetrahedron to the VGA monitor. The graphics manager encapsulates three modules, VGA_Controller, Bresenham_Receiver, and Framebuffer. It simply wires them together and does not contain any glue logic at the RTL level.

2.5.1 Description of Ports

- sys_clk (in, std_logic):
  - 25 MHz system clock
- draw_new_points (in, std_logic):
  - Tells the graphics manager to draw a new tetrahedron defined by the points given in packets
- packets (in, array_4x16_t):
  - Packet of 4 points to draw from the math manager. Each of the 4 row of the matrix is a separate point. The 8 MSBs of the row specify the x coordinate and the 8 LSBs of the row specify the y coordinate. The coordinate system has (0, 0) at the top left and (255, 255 at the bottom right).
- HS (out : std_logic):
  - Horizontal sync signal for the 640x480 VGA monitor. This is sent to the HS pin on the VGA connector through the on-board port.
- VS (out : std_logic):
  - Vertical sync signal for the monitor. This is sent to the VS pin on the VGA connector through the on-board port.
- red/green/blue (out, std_logic_vector(3 downto 0)):
  - 4-bits for each of red, green, and blue. They are sent to their respective pins on the VGA connector through the on-board port.
- done_drawing (out, std_logic):
  - Tells the central controller that the graphics manager has finished drawing the provided packet to the monitor
- ready_to_draw (out, std_logic):
  - Tells the central controller that the graphics system is back in idle and is ready to receive a new packet

2.5.2 RTL Diagram

At a high level, draw_new_points and packets are sent into the bresenham_receiver, which provides the framebuffer with the pixels to illuminate to form the 6 edges of a tetrahedron. The vga_controller counts horizontal and vertical pixels with each clock cycle, providing the pixel, HS, VS, and video_on signals to the framebuffer. The framebuffer precisely illuminates red, green, and blue at certain times to draw the tetrahedron. The framebuffer also directly outputs the HS, VS, done_drawing, and ready_to_draw signals. Each module will be explained in greater detail in the following sections.

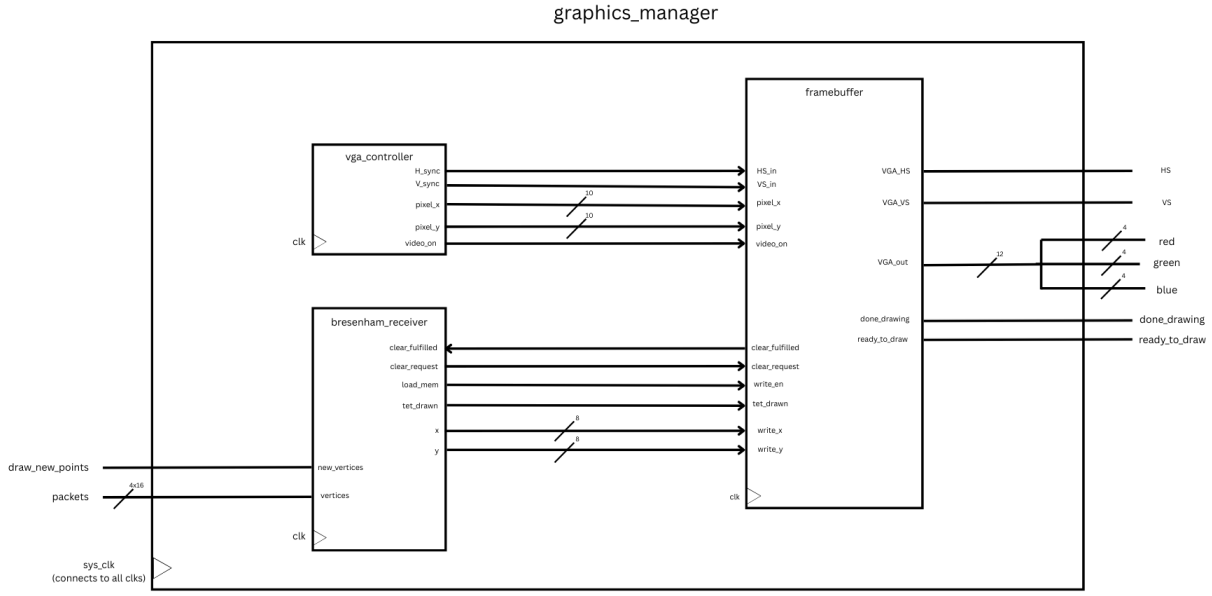

Figure 9: Graphics Manager Datapath

2.6 Parallel_Math (Math_Manager.vhd)

This module represents a midlevel shell that handles passing the correct args to four parallel instantiations of the update point component. A more detailed explanation for the math and algorithmic reasoning can be found in the the details for the Update Point (linalg_update.vhd) module.

2.6.1 Description of Ports

- clk_port (in : std_logic):
  - 25 MHZ clock signal divided down from 100 MHZ external FPGA Clock. Shared by all sub-components within this manager. For the sake of redundancy, we declare the port on the sub-components of this system but do not redefine this value.
- load_port (in : std_logic):
  - An enable signal that tells the manager to perform operations on the components that it currently reads on its input line.
- reset_port (in : std_logic):
  - Sends all sub-components that utilize a FSM back to their idle state which clears all registers, outputs, etc. by the next clock cycle.
- angles (in : array_2x16_t):
  - Two angles for which the points should be rotated by

- dirs (in : array_2x2_t):
  - The two axis that the points should be rotated about. Both dirs can equal the same axis. This ensures we always rotate about the same euclidean distance regardless if we only rotate about a single axis or two (Keeping the speed of rotation constant).
- points (in : array_4x3x24_t):
  - The four points each with x, y, z coordinate pairs that represent the tetrahedrons vertices position in local space (Not projected space).
- new_points (out : array_4x3x24_t):
  - The updated points post rotation that are returned back to the shell to be kept in memory.
- packets (out : array_4x16_t):
  - All four point packets projected onto our viewport in 2D. Since each coordinate $x_i, y_i \in [0, 255]$ we use 8 bits for each x, y. The 8 high bits of the vector are for x and the lower 8 are for y.
- set_port (out : std_logic):
  - Asserts that the output is ready to be communicated to the Graphics Manager once computation of the output packets is completed.

2.6.2 RTL Diagram

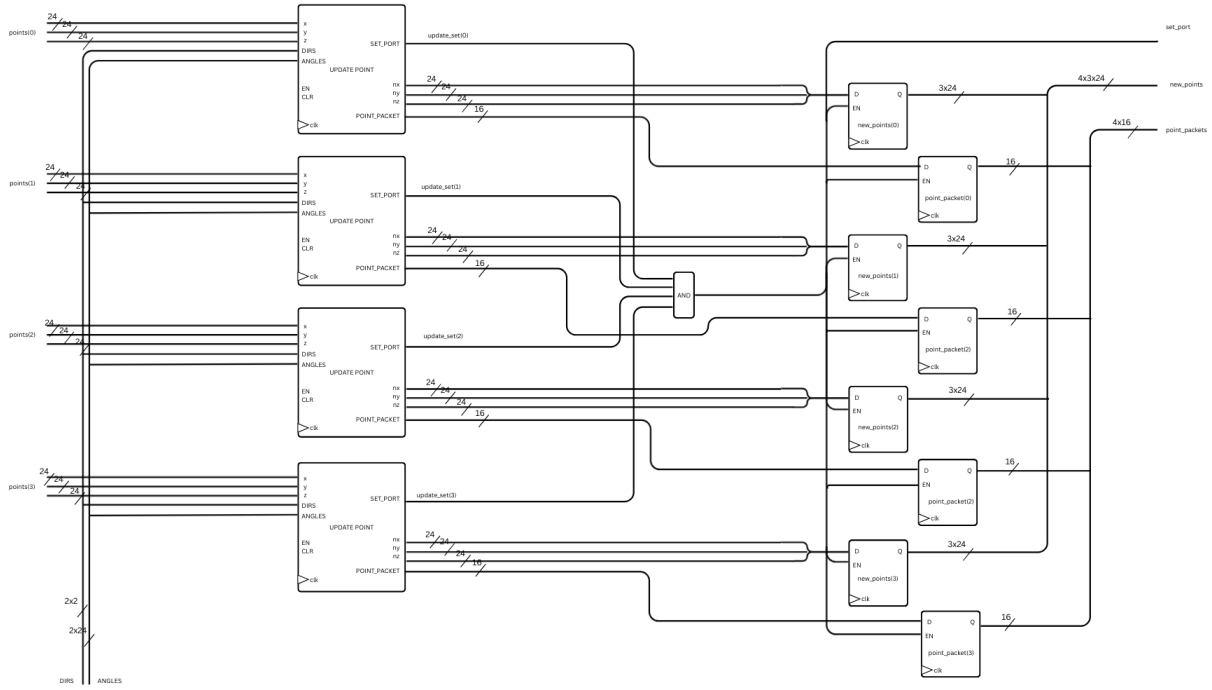

Figure 10: RTL Diagram showing Parallel Structure of Math Manager.

Each update_point module operates on a single point, multiplying by the rotation matrix and generating the projection from the rotated points. When the set_port for all four update_point modules are complete then the new points are latched to update the position of the tetrahedron and the point packets are latched to be sent to the Graphing Manager. Asserting reset_port results in all update_point modules being reset and thus their outputs being sent to zero.

2.7 Update_Point (Linalg_Update.vhd)

This is the main driver behind the Math Manager. This module aims to take in angles and axis for which each angle should be applied and rotate then project the points onto a 2D viewport to be sent to be drawn. It does so by instantiating two rotation modules and a projection module (Although the double instantiation of the rotation module could certainly be serialized, something we touch on in utilization). We want to spend a little bit of time motivating the specific algorithm we use to perform three matrix multiplications to get accurate/visually appealing 2D viewport coordinates (and the rotated 3D coordinates) however we utilize two different methods for computing the product for rotation and projection matrices respectively. We therefore define each individual algorithm in their respective module.

2.7.1 Description of Ports

clk_port (in : std_logic):

- Shared 25 MHZ
  - load_port (in : std_logic):
- Gated with set_ports of lookup tables and set_operands, must be asserted for a valid rotation matrix product to be asserted as ready
- reset_port (in : std_logic):
  - When asserted, clears all registered values and any values registered in memory or set_operands
- angle (in : array_2x16_t):
  - The two angles to points in the local space by. Both angles are in 4.12 unsigned fixed point notation with 12 fractional bits and 4 integer bits. Angles range from 0 to 2π and negative angles are expressed by $2\pi - \theta$.
- dir (in : array_2x2_t):
  - The two axis to apply the rotation on
- x, y, z (in : std_logic_vector(23 downto 0):
  - Current point from local space to update
- nx, ny, nz (out : std_logic_vector(23 downto 0):
  - New points after both rotations have been applied
- point_packet (out : std_logic_vector(15 downto 0):
  - Projected x, y, scaled, round, truncated, and intercepted into viewport. High 8 bits represent x and low bits represent y.

2.7.2 RTL Diagram

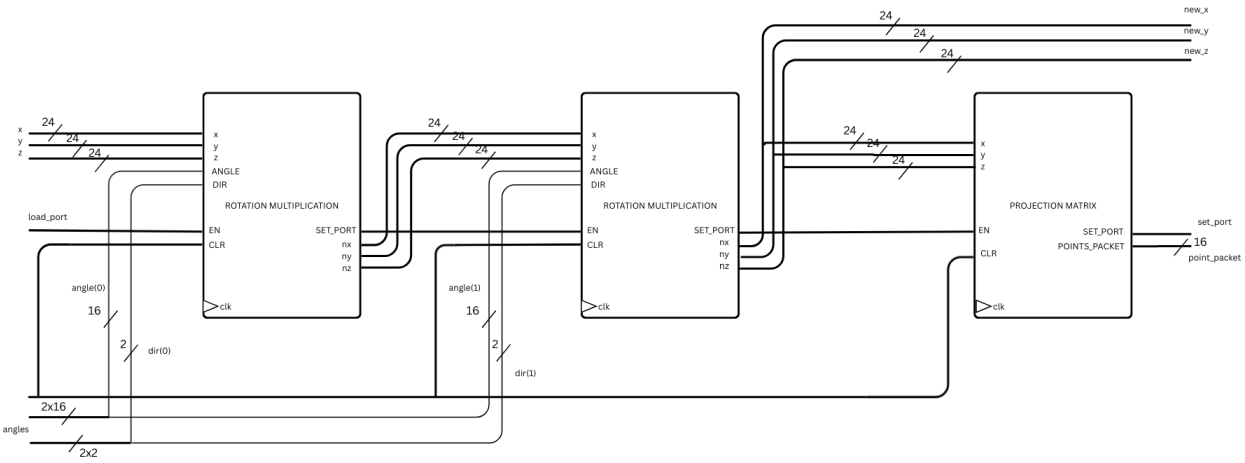

Figure 11: RTL Diagram showing process of two sequential rotations followed by multiplication by projection matrix.

At the assertion of load_port, new points on lines x, y, and z are rotated by the first rotation entity, the set port of the first module enables the second rotation matmul which then rotates the points output by the first rotation matmul by the second specified angle in the second direction. These points are the new points in local space and therefore are output by the datapath. Simultaneously they are sent to the projection matrix multiplication module which projects them into 2D space and outputs the point packet to also be picked up by the Math Manager. Since all three declared entities have synchronous resets, asserting reset_port just allows the entities themselves to handle clearling registers and subcomponents.

2.8 Rotation Matrix Multiplication (rotation.vhd)

We must perform matrix multiplication in the shape and order 3x3 times 3x1, where $R_{\hat{r}}(\theta)$ is the standard 3x3 rotation matrix about axis $\hat{r}$ (x, y, or z axis), left multiplied against $\vec{p}_i$, the point describing one of the four vertices of the tetrahedron in local space (Local space refers to where the points are in 3D, un-projected space and define their true position in Euclidean space). We use standard rotation matrices that are only well defined for rotations about the origin (As opposed to an arbitrary axis, requiring a 4x4 matrix and inhomogeneous term to be tracked in our points). The standard rotation matrices are

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}, R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}, R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Instead of defining this module for native matrix multiplication of this shape we instead leverage several facts about multiplication about the rotation matrix. First, the rows $(1,0,0),(0,1,0),(0,0,1)$ for rotation about each different axis preserve the coordinate of the axis we rotate along. Therefore, in our algorithm we save 3 multiplications by keeping the variable in the direction $\hat{r}$ specifies constant through multiplication. Second, the construction of the matrix means for coordinates not in axis of rotation we only need two products and an add. Additionally, the pattern these products must be computed in follows the same pattern ($o_1 \cos\theta - o_2 \sin\theta$ and $o_1 \sin\theta + o_2 \cos\theta$) for each matrix, therefore we only need a single module to assist in setting which values are multiplied by which trig function output. Overall this saves 4 multiplications and a decent amount of auxiliary control logic by limiting our module be specific for rotation matrix multiplication in the shape required.

2.8.1 Description of Ports

- clk_port (in : std_logic):
  - Shared 25 MHZ
- en_port (in : std_logic):
  - Gated with set_ports of lookup tables and set_operands, must be asserted for a valid rotation matrix product to be asserted as ready
- reset_port (in : std_logic):
  - When asserted, clears all registered values and any values registered in memory or set_operands
- angle (in : std_logic_vector(15 downto 0):
  - The angle for which we should address the lookup tables. The angle is in fixed point notation. It is unsigned with 4 bytes for the integer portion (thus $[0, 7]$) and 12 bytes for the fractional portion. Therefore a negative angle is actually indexed as $2\pi - \theta$.
- dir (in std_logic_vector(1 downto 0):
  - Encodes the axis corresponding the rotation matrix we want to multiply by.
- x, y, z (in : std_logic_vector(23 downto 0):
  - Current coordinates from single point in local space. Encoded as signed fixed point, with 11 integer bits and 12 fractional bits.
- nx, ny, nz (out : std_logic_vector(23 downto 0):
  - The new points after being multiplied
- set_port (in : std_logic):
  - Asserts when data is ready to other modules

2.8.2 RTL Diagram

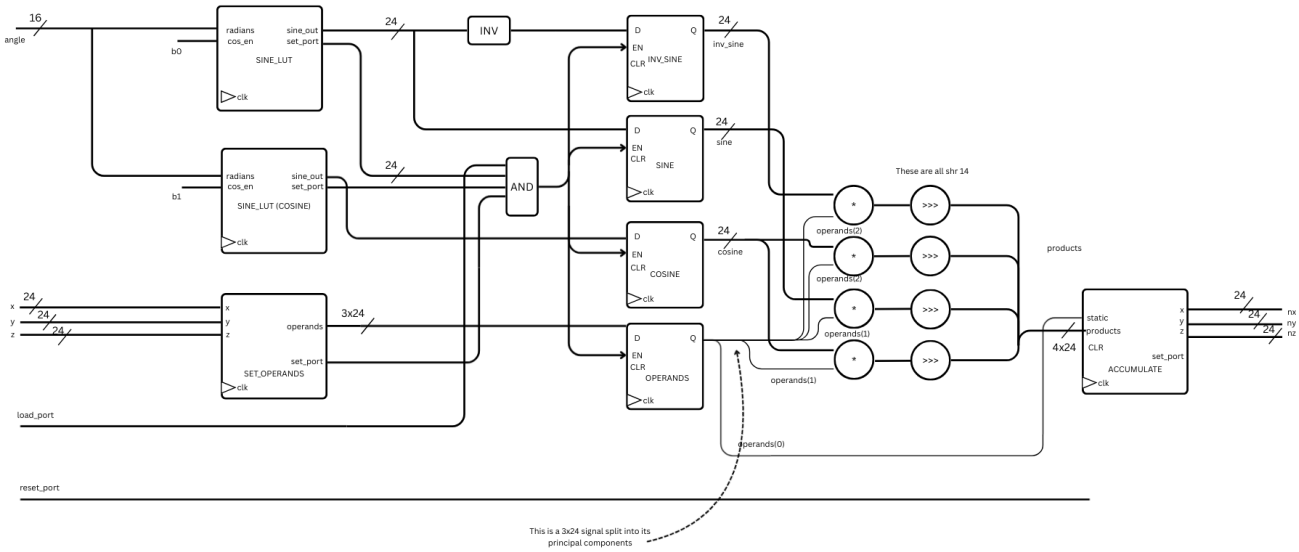

Figure 12: RTL Diagram for the rotation matrix multiplication datapath.

A single point broken into separate lines x, y, z comes into the datapath along with the angle and direction for the rotation. The angle is then used to hit the sine lookup table for the values of sine and cosine at that angle. The direction is used in set_operands, where the signal is decoded and used to set each operand into the correct signal relative to the axis we expect to rotate about. These values are not registered until the set_port for each subcomponent have been asserted. The operands 1 and 2 are then dotted as outlined by our algorithm and accumulated by the module accumulate_rotation, which then also sets the new values of x, y, and z depending on the encoded direction that we rotated about. Exactly one cycle after the set values are registered (Thus allowing the correct result to be computed by accumulate_rotation) we assert set_port, letting linalg_update know the data is ready.

2.8.3 Subcomponent A: Sine Lookup Table (sine_lut.vhd)

A little bit more complicated than simple ROM. The angle needs to be processed before it can address the table. Realistically we only need to store the value of cosine and sine for the angles we used, but technically this lookup table is robust to any angle, which helped in debugging due to having a range of angles we could use.

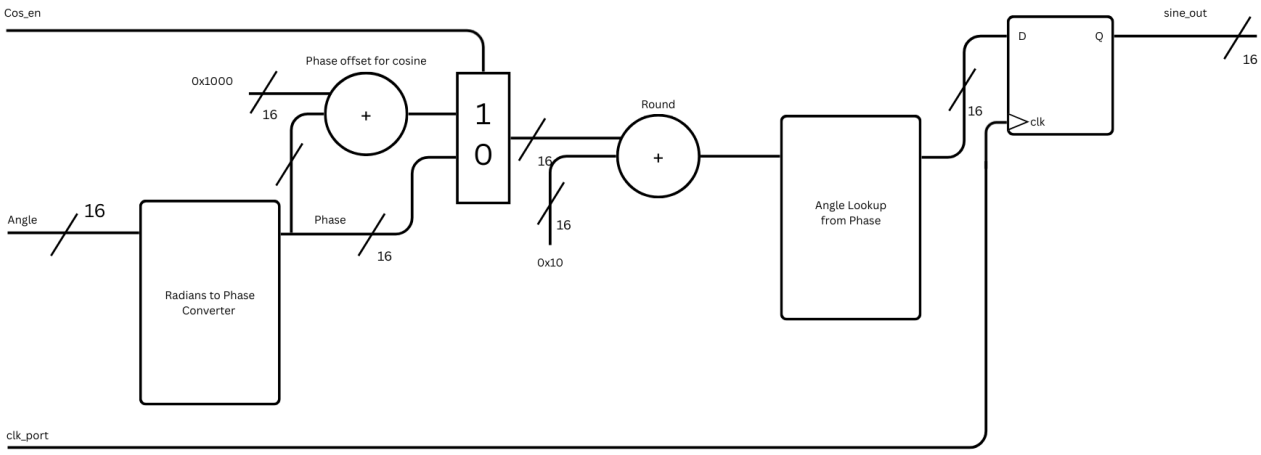

Figure 13: Datapath for Sine Lookup table.

The table itself is 1024 entries deep, 16 bits wide. The values within it are angles in signed fixed point notation 1.14, that is, a single integer bit and 14 fractional bits. The block radians to phase converter approximates a multiplication by $8/\pi$ to the input angle to convert our radians to a 16 bit unsigned phase to index the table. The lookup table has a port to enable a slight offset that adds $\pi/2$ converting the function to a cosine table. Before the phase indexes the table, we round by adding 16 (0x10) ensuring the phase is rounded off to a value witihin the lookup table.

2.8.4 Subcomponent B: Set Operands (set_operands.vhd)

Decodes direction directing x, y, and z signals into registers for operands 0, 1, and 2 depending on the direction we are multiplying by.

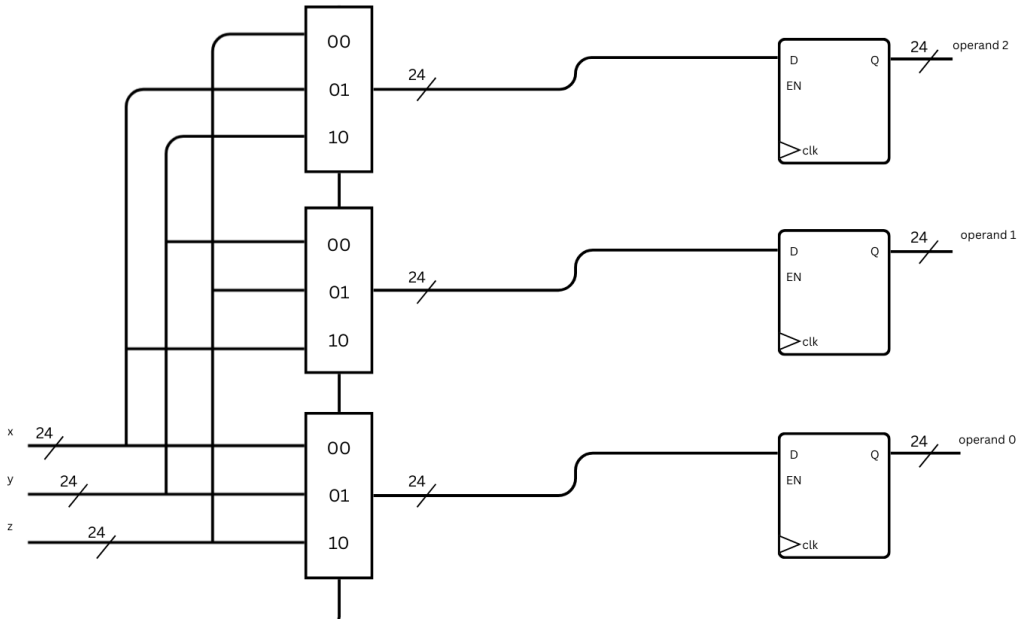

Figure 14: Datapath for setting operands to be used in rotation logic.

Simply decodes where x, y, and z should be latched from the dir signal.

2.8.5 Subcomponent C: Accumulate Rotation (accumulate_rotation.vhd)

Determines how to sum the four found products in main datapath depending on what axis we rotated about. The datapath just directs where the two sums and static variable should go for the given rotation

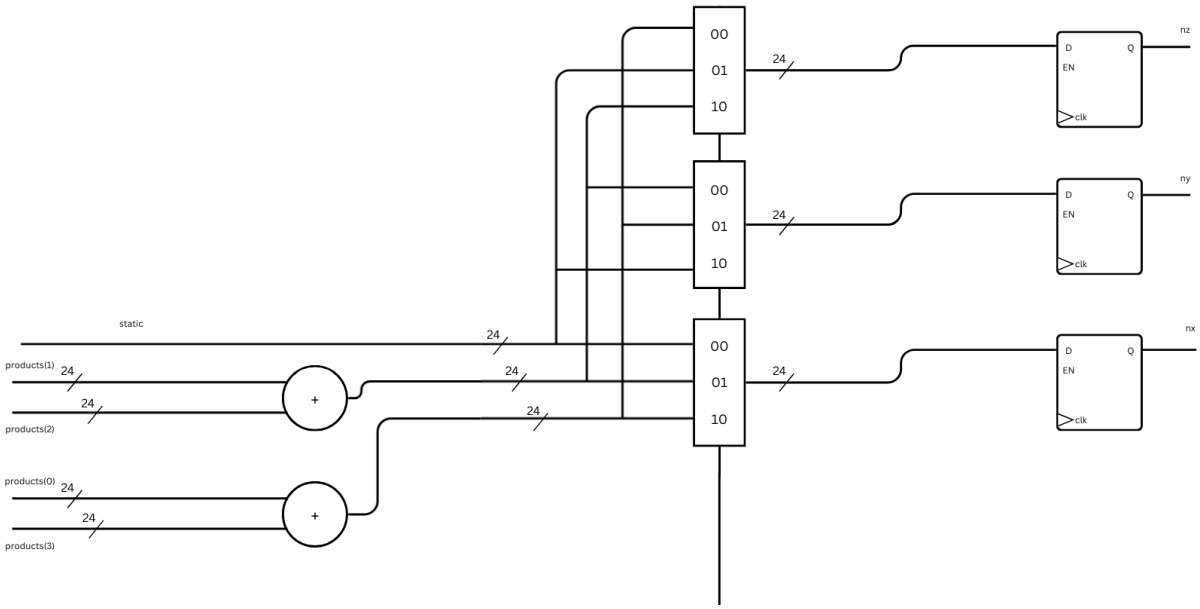

Figure 15: Accumulator datapath for summing dot products from matmul.

2.9 Projection Matrix Multiplication (projection.vhd)

The standard projection matrix is defined by,

$$P = \begin{pmatrix} m_{00} & 0 & 0 & 0 \\ 0 & m_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Where a is the aspect ratio and

$$m_{00} = \frac{\tan(FOV/2)^{-1}}{a}, m_{11} = \tan(FOV/2)^{-1}.$$

Normally the matrix would be a bit more complicated but since we neglect to draw faces we do not need the matrix to encode how faces lay on top of each other. Clearly computing m_{00} and m_{11} for continuous values of FOV would be incredibly expensive. From extensive testing on FOV values that worked, we found that an FOV of 70 provided the best depth while also avoid the issue of overflow in our framebuffer. The aspect ratio of our viewport was 1:1 since we plotted in a 256:256 space. Technically we could've opted for a 4:3 aspect ratio since the overall screen is 4:3, but the difference is up to visual choice. Therefore we let $m_{00} = m_{11} = \tan(FOV/2)^{-1}$. We precompute the value for some FOV and then load the values m_{00}, m_{11} as constant signals in 11.12 fixed point notation (signed, 11 integer, 12 fractional bits, for 70 FOV: 0x0016DA). Then the matrix multiplication $P\vec{p_i}$ is quite trivial. Notice that P is a 4x4 matrix so we need to include a homogeneous term 1 such that $\vec{p_i} = (x, y, z, 1)$. This is quite common in computer graphics, and is necessary to capture the depth of the points in our 2D project. Therefore for some $\vec{p_i}$, its projected value is $(m_{00}x, m_{11}y, 0, -z)$.

In order to convert this to the 2D points we want to draw, we need to extract the perspective information. Therefore we divide $m_{00}x$ and $m_{11}y$ by $-z$ to apply the perspective of points in space. These two quotients are then scaled, rounded, then packed into 16 bit signals as our coordinate pairs. Since we divide by $-z$ we need to avoid division by zero. A common fix to set some tolerance for which any $z < \epsilon$ forces $z = \epsilon$. In general ϵ is usually quite small but in the context of computer graphics, especially for a small viewport like ours, we need ϵ to be large to limit the perspective of objects as they get close to $z = 0$. For our projection matrix we settled on setting $\epsilon = 8$ which we found qualitatively, checking values until we got one that mitigated this issue well. Finally we need to arbitrarily scale then intercept the resulting two points by 128 to push it into the viewport. Since our local space must be centered around the origin this allows negative values of x, y, and z. Therefore, we must take our values, scale them by a power of 2 to make the shape larger visually (a cosmetic decision) then add 128 to each point to recenter about (128, 128) on our viewport. Hence we break the matrix multiplication into a much simpler datapath by leveraging the projection matrices properties.

2.9.1 Description of Ports

- clk_port (in : std_logic):
  - 25 MHZ Shared.
- load_port (in : std_logic):
  - Enables the projection math to project the available points onto our viewport.
- reset_port (in : std_logic):
  - Clears all registers and subcomponents when asserted
- x, y, z (in std_logic_vector(23 downto 0):
  - Current points in local space to project onto viewport, in 11.12 signed fixed point notation.
- point_packet (out : std_logic_vector(15 downto 0):
  - Points to be drawn projected onto viewport
- set_port (out : std_logic):
  - Asserts that data is ready to be ready from output.

2.9.2 RTL Diagram

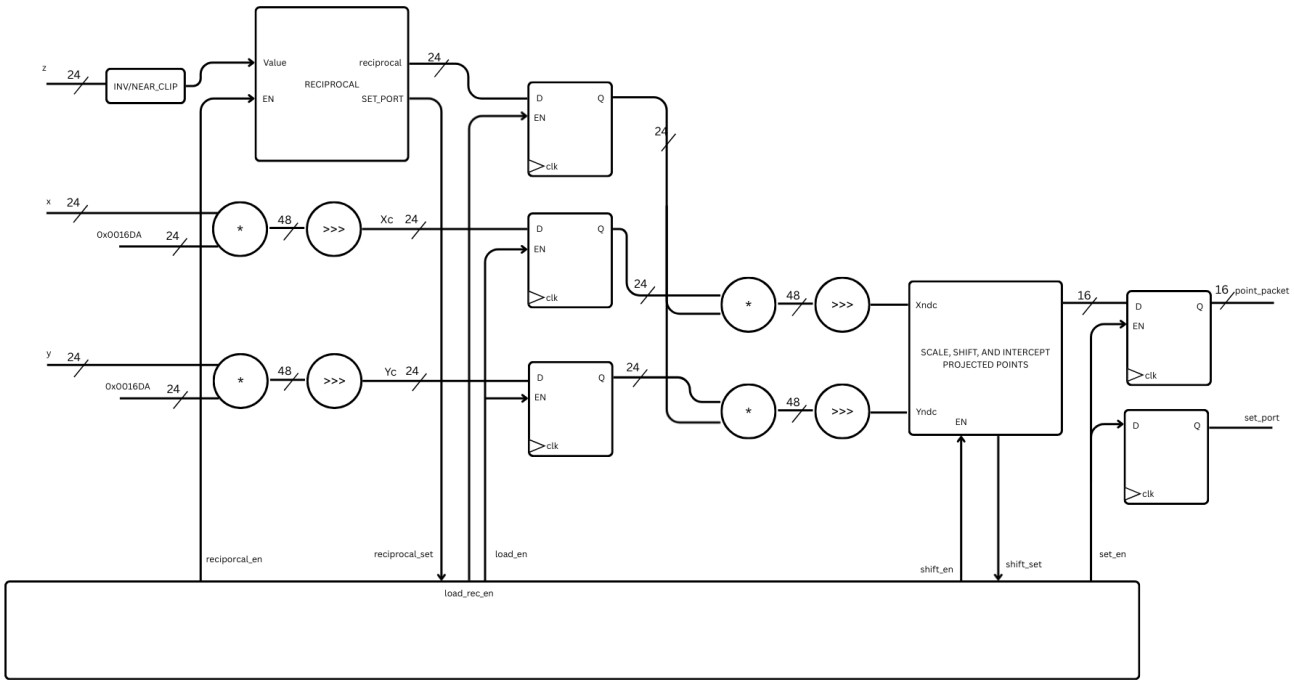

Figure 16: RTL Diagram for the projection matrix multiplication datapath and controller.

The input signal z is immediately inverted and simultaneously checked if less than our near clip value of 8. The correct value for z is then loaded into the reciprocal entity and waits to be enabled (See 2.10 for Reciprocal and its subcomponents). While this occurs, x and y are multiplied against m_{00} and m_{11} respectively, shifted back to 11.12 and latched when load_en is asserted. After these values are latched, reciprocal_en is asserted, the reciprocal $1/-z$ is computed and Reciprocal's set_port is asserted communicated to the FSM that it can be latched. Once our products and reciporcal are computed, shift_en is high, allowing for the divided values (x_{ndc}, y_{ndc}) (our unscaled x,y with perspective) to be scaled (x16), rounded to the nearest integer and intercepted with $b = 128$. Once shifting has completed, shift_set is asserted, allowing the controller to assert set_en which latched the point packet and high set_port to notify the Math Manager that the packed points are ready to be read.

2.9.3 FSM Diagram

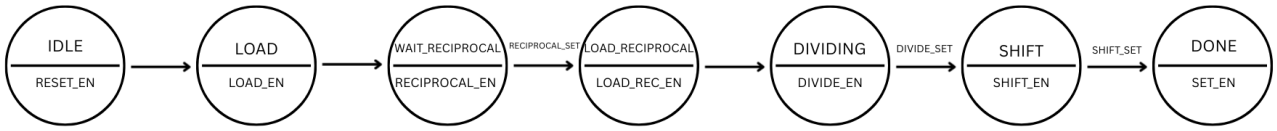

Figure 17: FSM for projection matrix multiplication.

The FSM is very simple, if there is no signal specified as transitioning the FSM then it transitions on the next clock cycle. If at any point reset_port is asserted the FSM transitions to IDLE which asserts reset_en, clearing the registers and used entities.

2.10 Reciprocal (reciprocal.vhd)

We must divide $(m_{00}x, m_{11}y)$ by $-z$, otherwise our projection will look flat and uninteresting. Therefore we must implement a divider. We felt that the simplest way to do this would be for some arbitrary a, find $1/a$ and multiply by the reciprocal.Then our implementation should, for some value a, fetch a guess from ROM then iteratively improve it and return to the caller. Since we want the number of iterations to be low, we want the initial guess to be a good as possible. Therefore we define a normalization process such that all arbitrary values we input can be normalized into a similar form, and we can provide the reciprocals for a range of values in this form. Therefore we normalize the value to be in the range $[1, 2)$ in signed fixed point notation 1.23 using the following process. Since value is always 11.12 fixed point, we compute the distance from the 1st high bit to MSB. Shifting left by this value pushes our value into our expected range. This allows us to index with the fractional bits (We use the first 10 fractional bits to limit the depth of our ROM) and get a high resolution of guesses that are relatively close to the real reciprocal. If we reframe this problem differently, we can leverage tools from numeric analysis to derive a way of iteratively finding $1/a$ given our seed value. We then define the function,

$$f(x) = \frac{1}{x} - a.$$

Where a is our value we want the reciprocal of. Clearly a root of this function is $1/a$. From newton's method we then derive an algorithm for the next guess,

$$x_{n+1} = x_n(2 - mx_n).$$

Where m is the normalized value in range $[1, 2)$ and x_0 is our seed. Once we iterate with newton's method, we can de-normalize by keeping track of how far we had to shift our initial value left by to get it in the range our table expects. This number is then our reciprocal.

2.10.1 Description of Ports

- clk_port (in : std_logic):
  - 25 MHZ Shared.
- load_port (in : std_logic):
  - Enables the projection math to project the available points onto our viewport.
- reset_port (in : std_logic):
  - Clears all registers and subcomponents when asserted
- value (in : std_logic_vector(23 downto 0):
  - Value to get reciprocal for, in 11.12 signed fixed point notation.
- reciprocal (out : std_logic_vector(23 downto 0):
  - Reciprocal of value in 11.12 signed fixed point notation. Is not yet un-normalized however. That is the job of the caller (projection in this instance).
- set_port (out : std_logic):
  - Asserts when reciprocal is ready to be read.

2.10.2 RTL Diagram

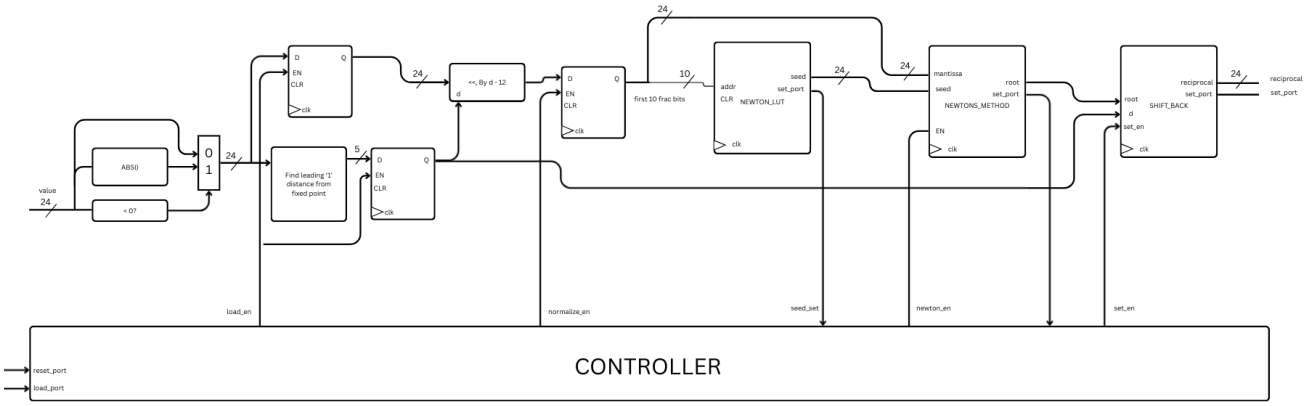

Figure 18: RTL Diagram for reciprocal datapath.

We take the absolute value of the incoming value (Storing whether it was negative, not pictured), latching it once set_port is enabled (thus transitioning FSM into load, asserting load_en). Simultaneously, logic determines the distance from the fixed point, thus allowing us to calculate and latch the distance that we must shift left (and later right) to get our normalized value. Once load_en is asserted, we shift the value getting a normalized mantissa, we index the seed lookup table, call our newtons_method subcomponent and wait for it to assert completed. We then take the result as well as the distance we normalized by, de-normalize, reapply the sign bit (Again not pictured) and assert set_port to denote that the reciprocal is ready to be read by the projection module.

2.10.3 FSM Diagram

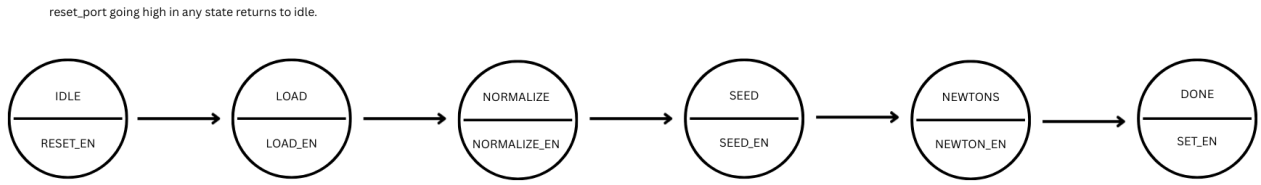

Figure 19: FSM Controller for Reciprocal.

Again, a very simple FSM that essentially handles the logic for waiting for subcomponents and external modules to finish computing something that isn't well defined in number of cycles. reset_port being asserted at any point results in a transition back to IDLE. Note, the transitions for seed_set and newton_set are not pictured but transition the FSM to NEWTONS and DONE respectively.

2.10.4 Description of Memory

See Subcomponent A: Newton LUT, for short description of memory

2.10.5 Subcomponent A: Seed/Guess ROM (newton_lut.vhd)

Simple interface over a 1024 deep, 24 bit wide ROM that is indexed with the first 10 fractional bits of the normalized mantissa. Returns the closest reciprocal "guess". The guess is a 6.17 signed fixed point value. We provide extra fractional bits and less integer bits since its the expected reciprocal for a normalized value, hence we would prefer the extra precision in this instance.

2.10.6 Subcomponent B: Newton's Method

Performs the outlined algorithm for two iterations given a mantissa and initial guess. We must be slightly careful since the second iteration, x_1 is in 11.12 fixed point notation, not 6.17 like $x_0 \sim$ seed.

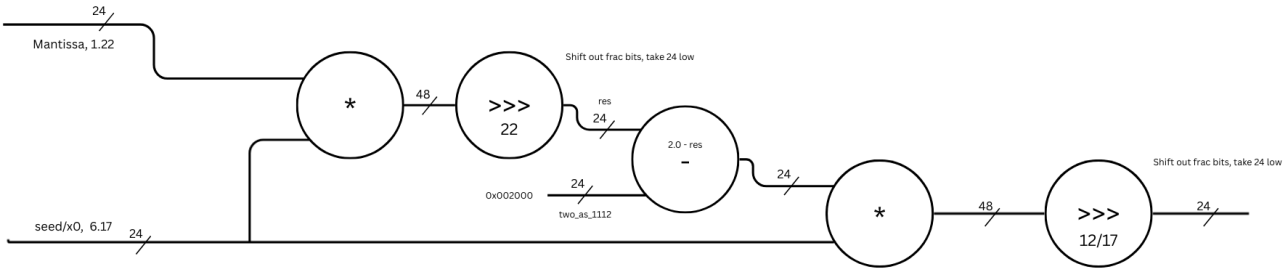

Figure 20: A single step of newtons method.

The first figure defines the arithmetic (without pipelining) performed with some x_n and mantissa m.

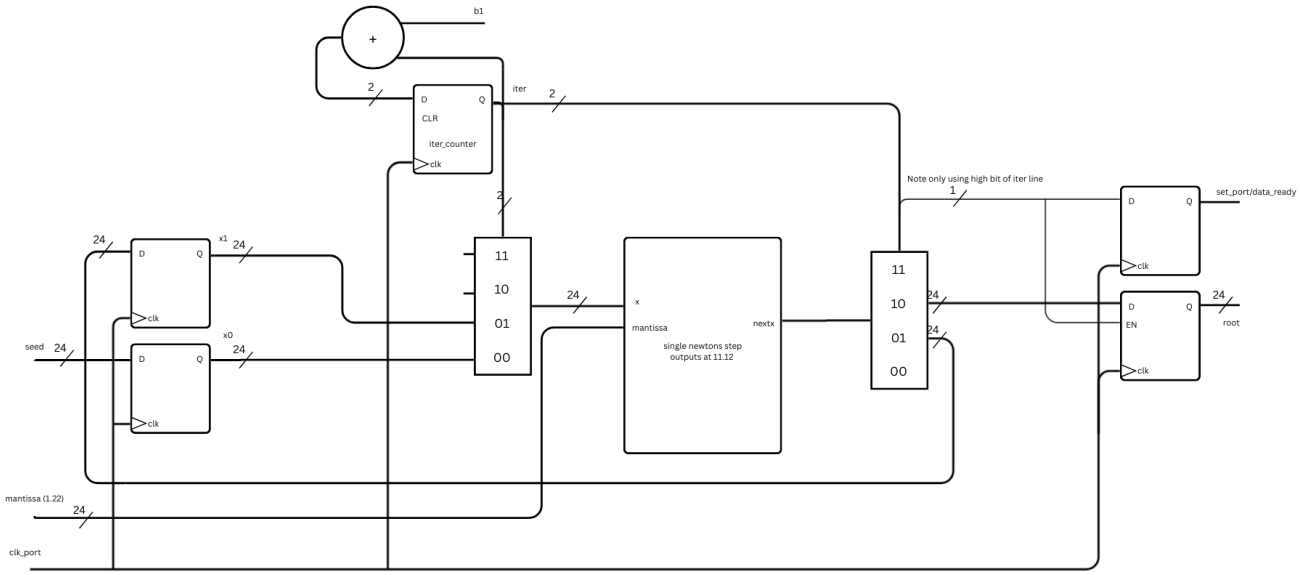

Figure 21: A single step of newtons method.

The datapath figure just outlines how for each count we must decode which input to pick based on our iter_counter and where to send the output of a single step. Once x_2 is computed we can latch the value and assert set_port to notify the reciprocal module that x_2 is ready to be read.

2.11 Angle_Dir_LUT (angle_dir_lut.vhd)

2.11.1 Description of Ports

- clk (in, std_logic):
  - 25 MHz system clock
- addr (in, std_logic_vector(7 downto 0)):
  - ASCII code to address in the ASCII_table from the UART_Receiver when not initializing. When initializing, addr is all 0s which maps to a special index.
- request (in, std_logic):
  - Signals to the LUT that there addr is valid and to provide the angles and dirs
- read_angles (out, array_2x16_t):
  - Based on the addr, provides the corresponding angle for that key press. Datatype is what math manager uses as input for angle.
- read_dirs (out, array_2x2_t):
  - Based on the addr, provide the corresponding direction for that key press. Datatype is what math manager uses as input for direction.
- lut_valid (out, std_logic):
  - Signals that the outpts angles and directions are valid to be read
- lut_invalid (out, std_logic):
  - Signals that the key press was not mapped, so nothing should be done
- reset_press (out, std_logic):
  - Signals that the R key was pressed, which is a special case for the math manager

2.11.2 RTL Diagram

The module receives a request signal when the 8 bit address is valid. It uses that address to index a ROM table mapping 8 bit ASCII codes to unique indices. These indices are used to access the direction_table and angles_table, which provide the angles and direction for that keyboard input. The set_data process, which is synchronous, decides the lut_valid, lut_invalid and reset_press signals based on the index from the ASCII table.

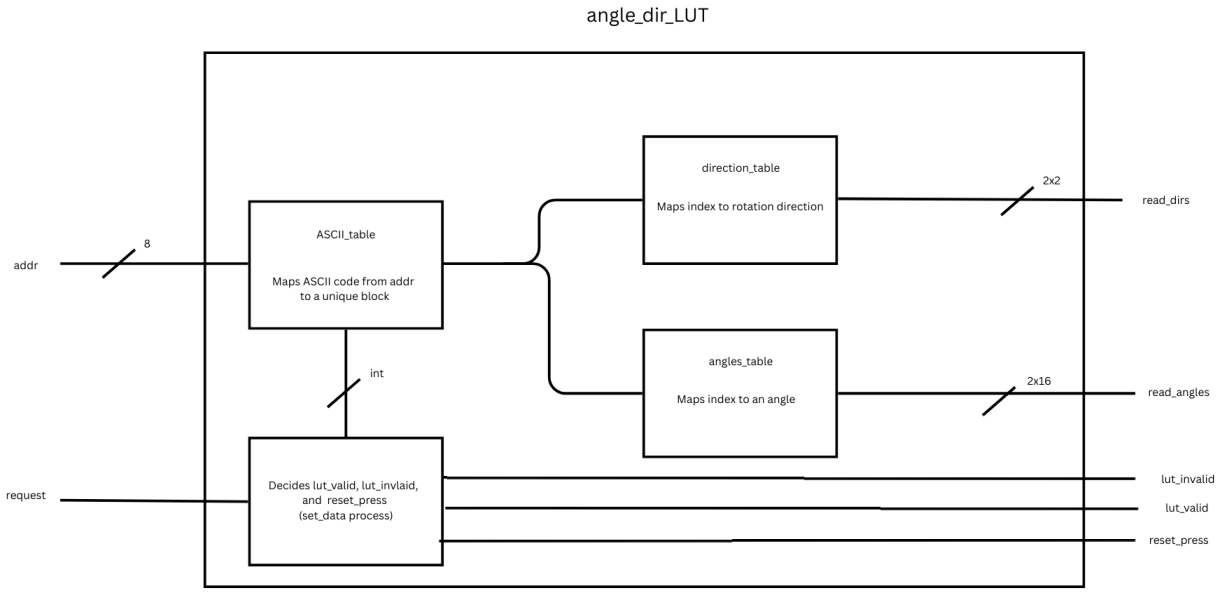

Figure 22: Angle Direction LUT RTL

2.11.3 Description of Memory

Idx ranges from 0 to 14, requiring 4 bits. All 3 tables are inferred as ROM. The ASCII_table has width 4 and depth 256. The direction_table has width 4 and depth 4. The angles_table has width 32 and depth 4.

2.12 UART_Receiver

Receives the UART ASCII code relayed by PuTTY from a keyboard button press. When data_valid output goes high, the 8 bits of data are valid to be read. It then consumes the stop bit before being ready to read another character.

2.12.1 Description of Ports

- clk (in, std_logic):
  - 25 MHz system clock
- rx (in, std_logic):
  - Serial input from the USB-UART bridge. Uses a 9600 baud rate to read a start bit (0), 8 data bits (LSB first), and a stop bit (1) , with idle bits (1) in between each code.
- data (out, std_logic_vector(7 downto 0)):
  - 8 bits of data that was received.
- data_valid (out, std_logic):
  - Signals whether the data present on the data output is valid to be read.

2.12.2 RTL Diagram

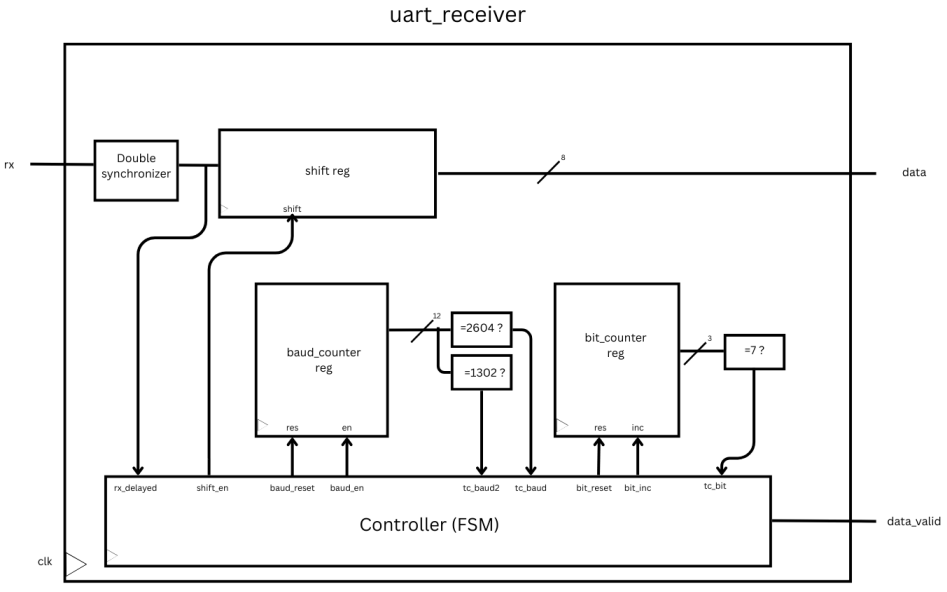

Figure 23: UART Receiver Datapath

The double synchronizer is a precaution to avoid metastability by ensuring that rx is synchronized with the clock. This is important with user input because the button press can occur at any time in the system clock cycle. When shift_en is asserted, the shift register performs a right shift because the LSB of the data is sent first. When baud_en is asserted, the baud_counter counts up to 2604, the approximate ratio of the system clock rate (25 MHz) to the baud rate (9.6 kHz), before resetting to 0. This tells the controller that a baud period has passed and that a new bit should be shifted in. On the first shift, it only counts up to 1302 to ensure that rx is sampled in the center of each bit. After shifting in a bit, the bit_counter register is incremented so that the controller knows when data_valid should be asserted.

2.12.3 FSM Diagram

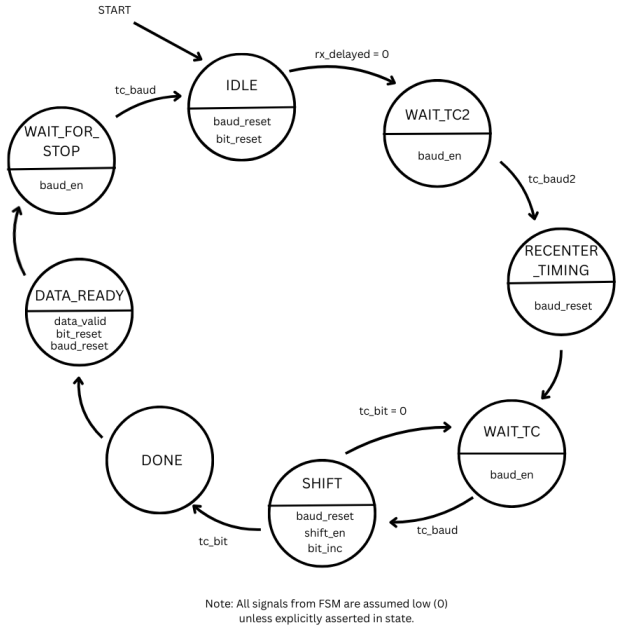

Figure 24: UART Receiver FSM

The receiver starts in the IDLE state, resetting the baud and bit registers. when rx goes low, it enables the baud counter and waits for half of the baud period. Then, in RECENTER_TIMING, it resets the baud register to prepare for shifting data in. The FSM then transitions between WAIT_TC and SHIFT, sampling rx 8 times and shifting these values into the shift register. When 8 values have been shifted in, it transitions to the DONE state for one cycle before going to DATA_READY, where bit_valid is asserted and the counters are cleared. Finally, in WAIT_FOR_STOP, it waits for a final baud period to "consume" the stop bit before returning to IDLE where the cycle can be repeated.

2.13 Central_Controller (Top Level Component)

While serving as the shell for the project, the central_controller also contains an FSM which coordinates the interaction between all the modules. Its control signals allow it to make decisions about when data from the keyboard is ready, when how to load the math manager, and when to tell the graphics manager to draw a new tetrahedron to the monitor.

2.13.1 Description of Ports

See sec. 1.3

2.13.2 RTL Diagram

The address process in the central controller requests an angle/direction from the angle_dir_lut while also deciding what the addr for the LUT should be. It distinguishes between two cases: when init_control is asserted, it provides and addr of 00000000, which is mapped to a rotation of 0 degrees. This is useful for initialization, when the math manager is solely used to project the starting points, not to rotate them. When map_press_control is asserted, the address process connects the LUT addr to the data (ASCII code) from the uart_receiver.

The currpoints process decides what should be stored in the current_points register, which contains the most recent set of 3D points. when init_control is asserted, it simply sets curr_points to the starting points, which create an equilateral tetrahedron centered at the (0,0,0). The starting points are the following: (26, 26, -26), (26, -26, 26), (-26, 26, 26), and (-26, -26, -26). The other case is when set_port (from math manager) is asserted, signaling that math has been completed. In this case, the current_points register is updated to hold the new_points provided by math. Points, the input to the math manager, is always tied to the output of the current_points register.

Finally, the datapath contains two other components: the packets register and math_wait register. The packets register's purpose is to keep the packets input to the graphics_manager stable; packets is only high briefly when the math manager finishes. It loads a packet from math when math is finished (set_port high). The math_wait register allows the central controller to wait in the MATH_WAIT state for 20 cycles (explained below).

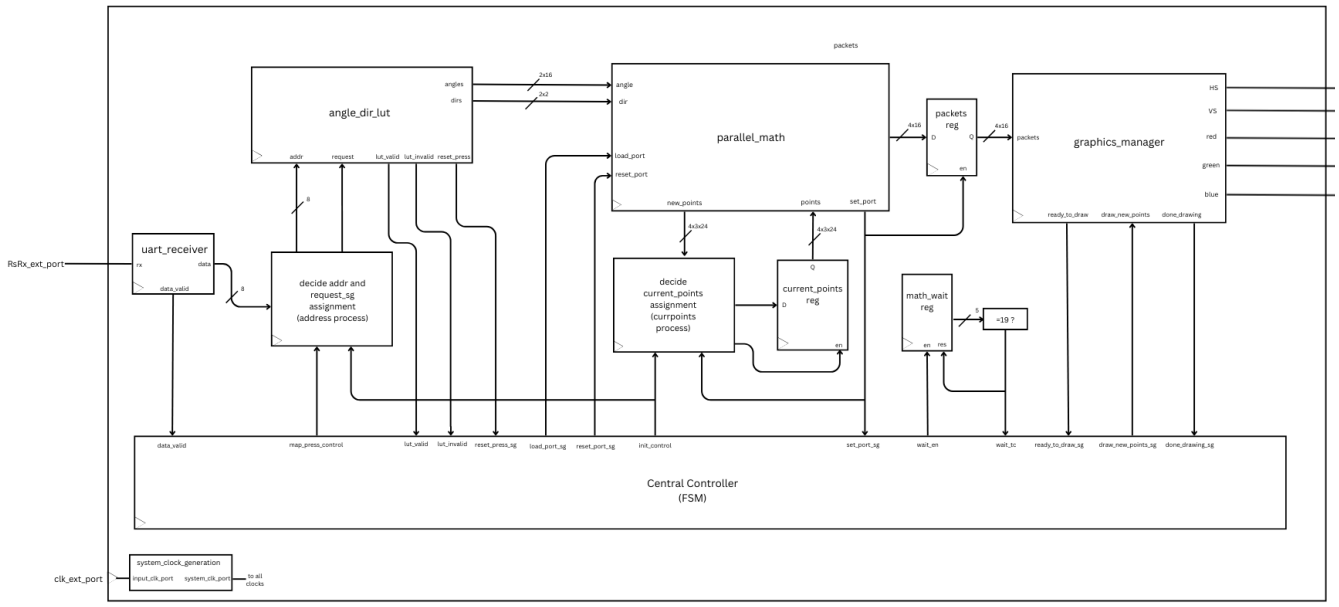

Figure 25: Central Controller Datapath

2.13.3 FSM Diagram

The FSM starts in the INIT state, outputting init_control for the processes explained above. When the LUT output is valid, it transitions directly to MATH, where the math module is activated (reset_port deasserted, load_port asserted). When MATH finishes, the FSM waits in WAIT_SCREEN for the graphics manager to be ready to draw before transitioning to START_DRAW. Here, the graphics manager is activates and draws to the screen in the DRAW state. When it is finished drawing, it enters the IDLE state for the first time and waits for data from the uart_receiver. Upon valid data, it goes to MAP_PRESS, asserting map_press_control for the standalone processes. If the LUT is invalid (key pressed is not valid), it transitions back to the IDLE state to wait for a new key press. If the LUT is valid, but the special flag, reset_press is asserted, the FSM transitions back to INIT to redraw the original tetrahedron. Finally, for a valid rotation key press (single: WASDQE, double: IJKLUO), it transitions to the WAIT_MATH state. We added this state becasue we discovered that the math manager requires reset_port to be low for a period of time before load_port is enabled. After this waiting period (20 cycles), the FSM transitions to MATH, and the cycle is the identical to the initialization.

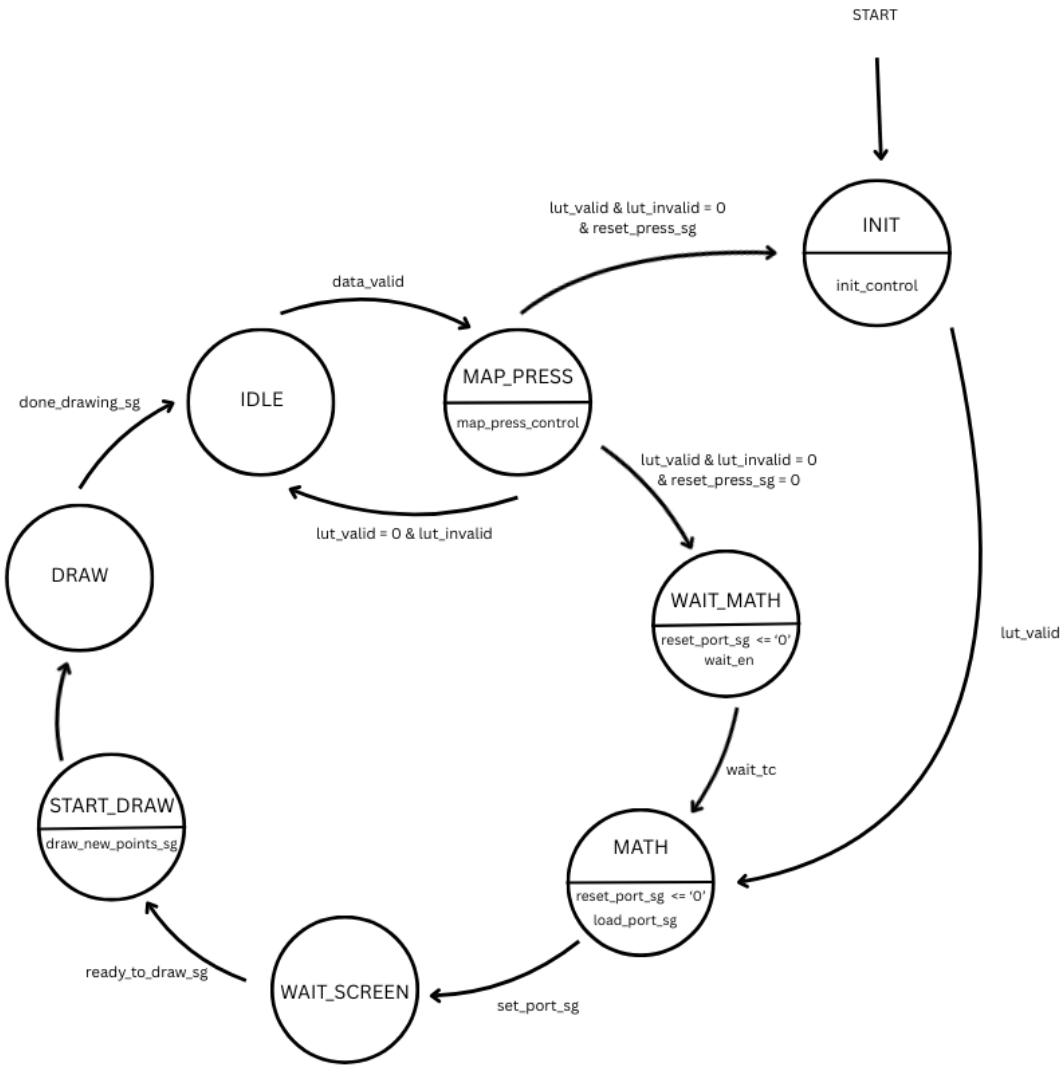

Figure 26: Central Controller FSM

3 Design Validation

3.1 VGA_Controller (vga_controller.vhd)

I followed the old Lab 5 that was posted on Canvas which provides broad instructions for how to create the VGA_controller module. However, since we were running the whole system on a 25 MHz clock, it was even easier because we did not need to divide the system clock down to match the VGA clock frequency. Simulation was useful for verifying that HS and VS were asserting at the correct pixel_x and pixel_y values. However, it was useless for determining of the color-bar test was correct. For this, we coded a shell with the controller and provided color-bar test module. From this, we was able to verify that my controller timing and color assignments are correct. For this part, we considered it a success when the color-bar worked correctly without glitching, flickering, etc.

3.1.1 Behavioral Simulations

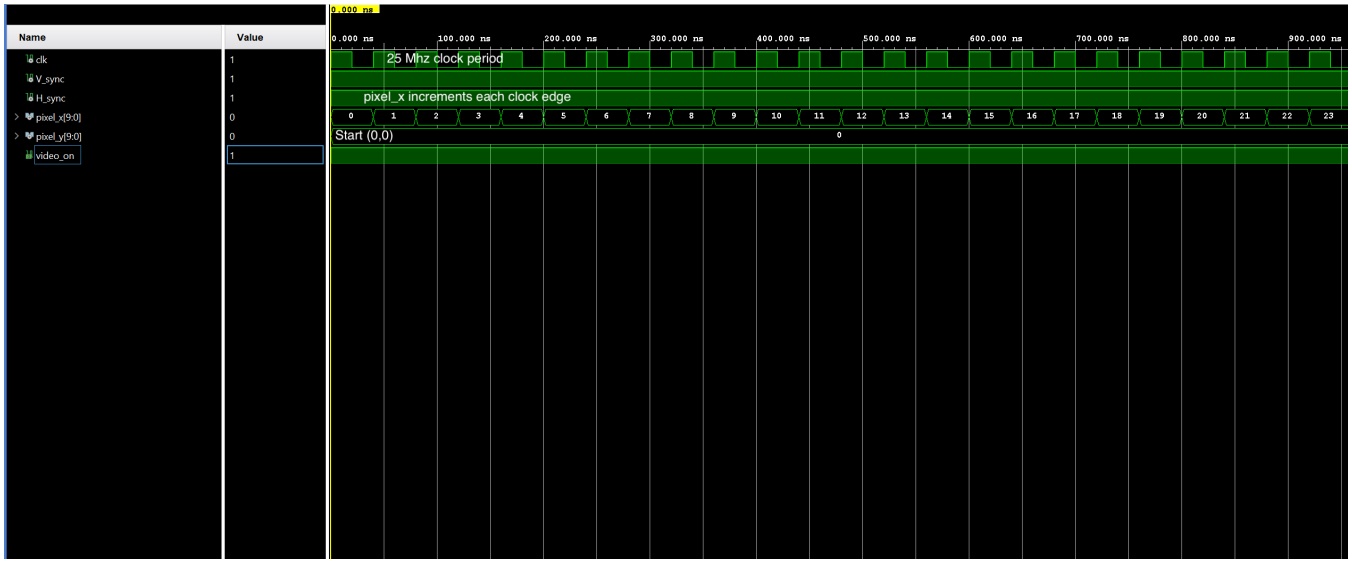

Figure 27: VGA Controller Simulation (Image 1)

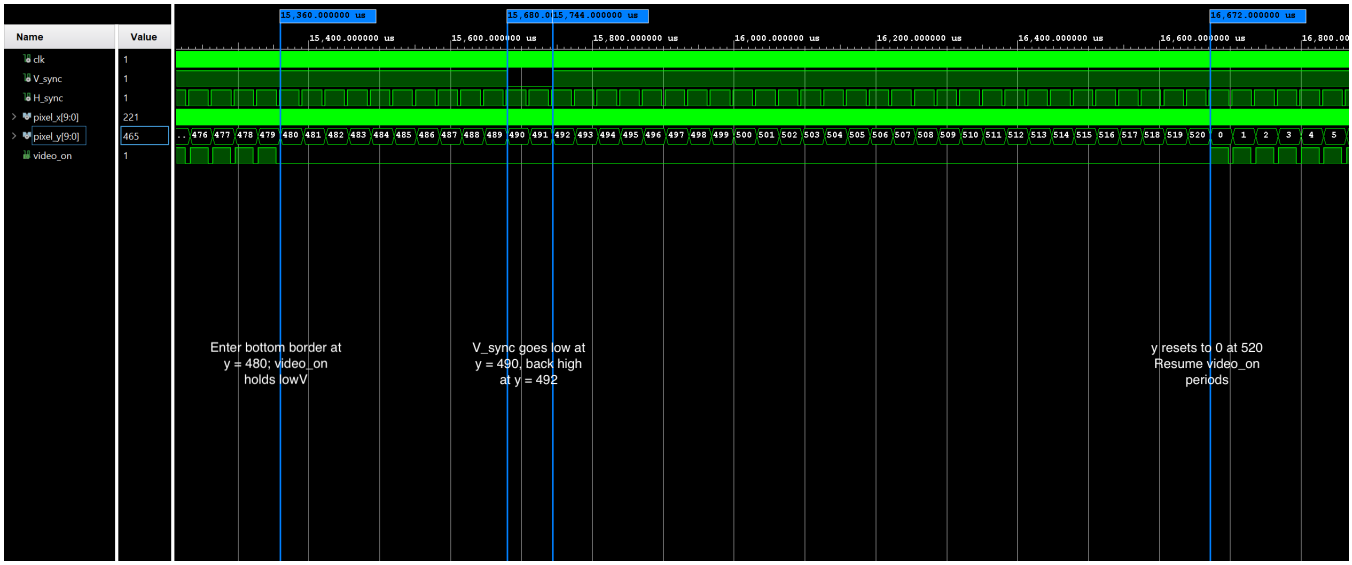

Figure 28: VGA Controller Simulation (Image 2)

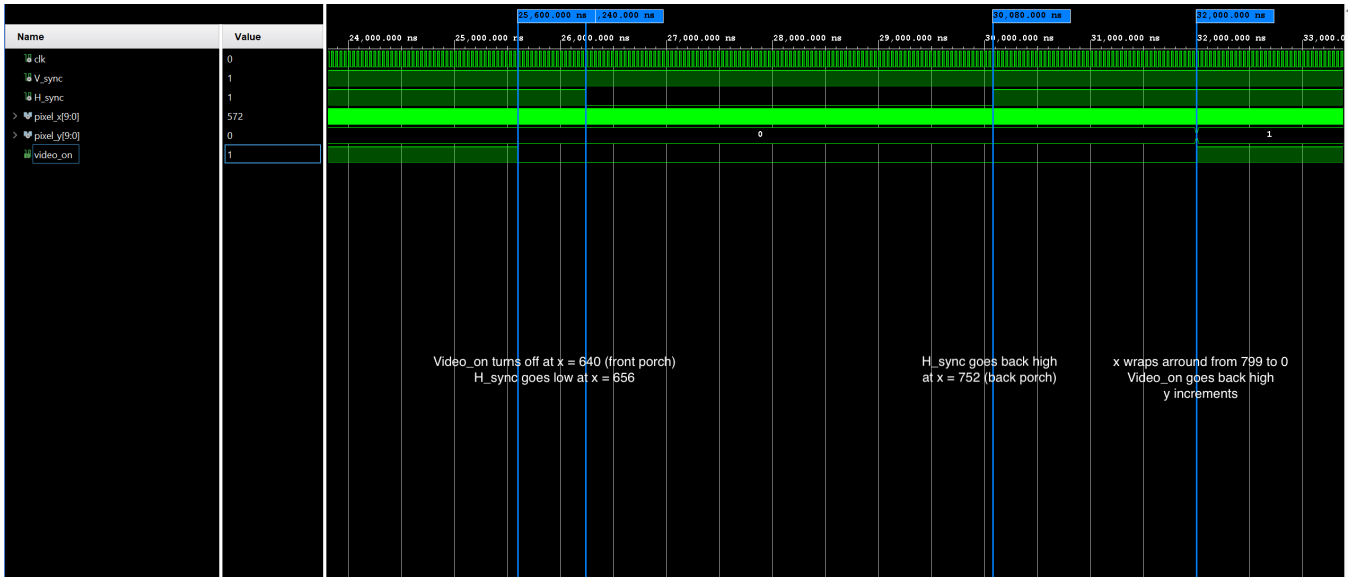

Figure 29: VGA Controller Simulation (Image 3)

3.1.2 Hardware Validation

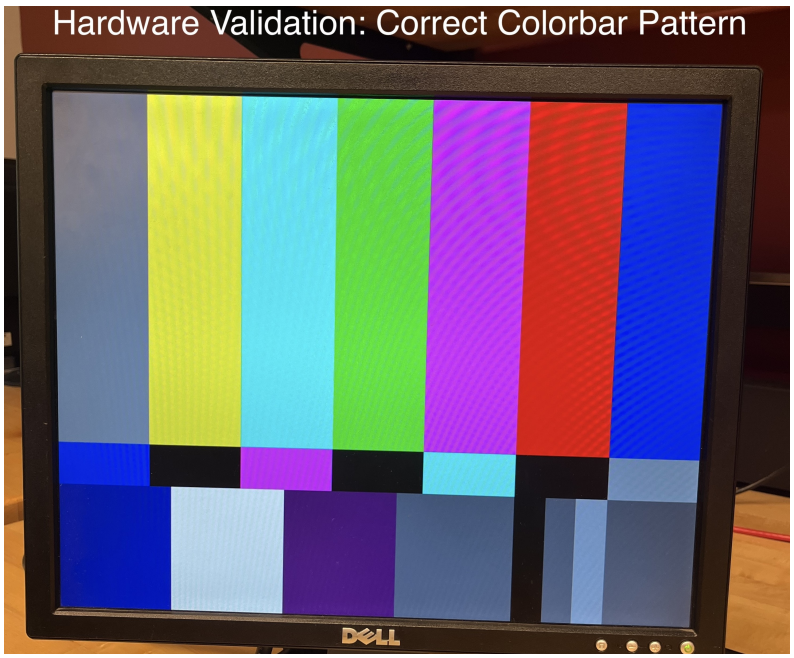

Figure 30: VGA Controller Hardware Validation

3.2 Bresenham (bresenham.vhd)

Because this module was directly transcribed from Edwards, my goal was to replicate the timing diagram that was provided in his lecture notes. We coded a testbench with the same endpoints, (0,0) and (7,4), as the notes. The simulation immediately matched the notes, so we moved on to other components. Note that the simulation was done with a 100 MHz clock before we decided to use a 25 MHz clock, but the functionality is not affected by the clock speed. We did not do a direct hardware validation for this component, instead opting to test it as part of a subsystem and discover errors along the way. The only bug we found was that in hardware, for nearly-horizontal lines, the algorithm would not draw the line correctly, but drew a slope that was too steep and wrapped all the way around the framebuffer to the endpoints. This was easily fixed by widening the err and e2 variables' bit width; the low slope resulted in overflowing err and e2 variables. After fixing this bug, the module worked for all cases we tested in hardware.

3.2.1 Behavioral Simulations

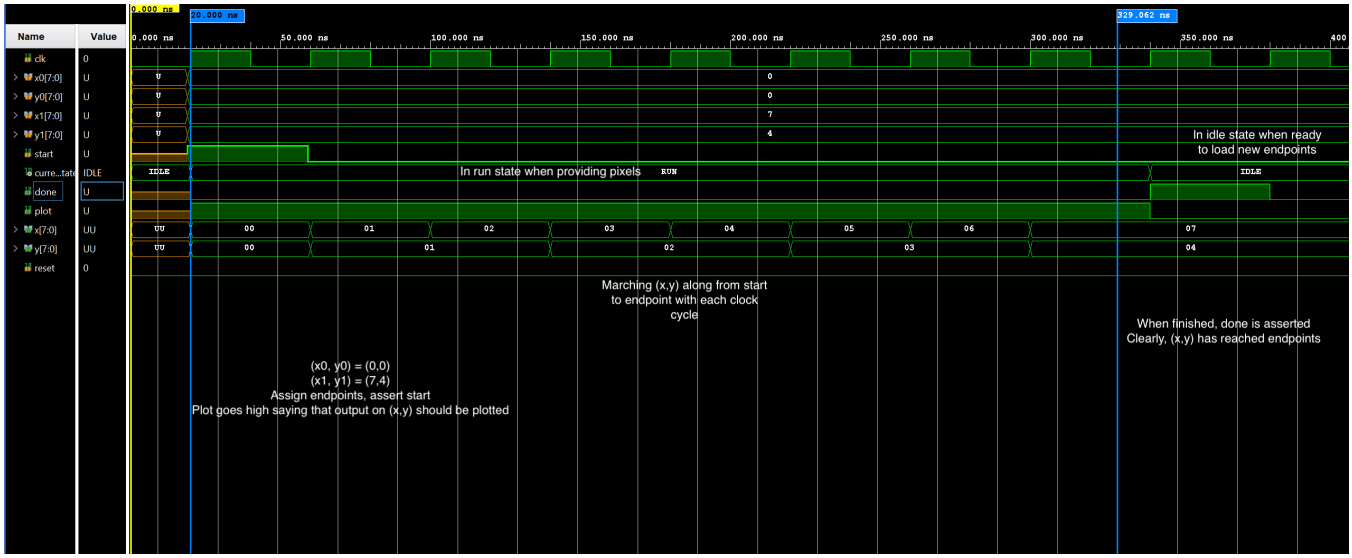

Figure 31: Bresenham Simulation (Image 1)

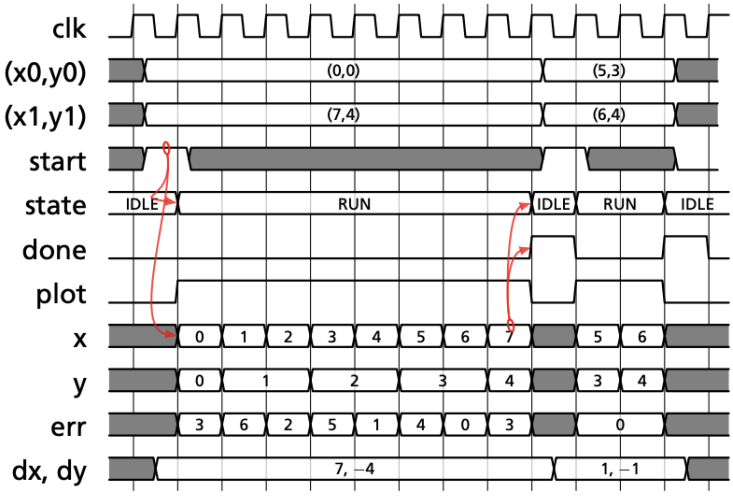

Figure 32: Edwards Lecture Screenshot Comparison

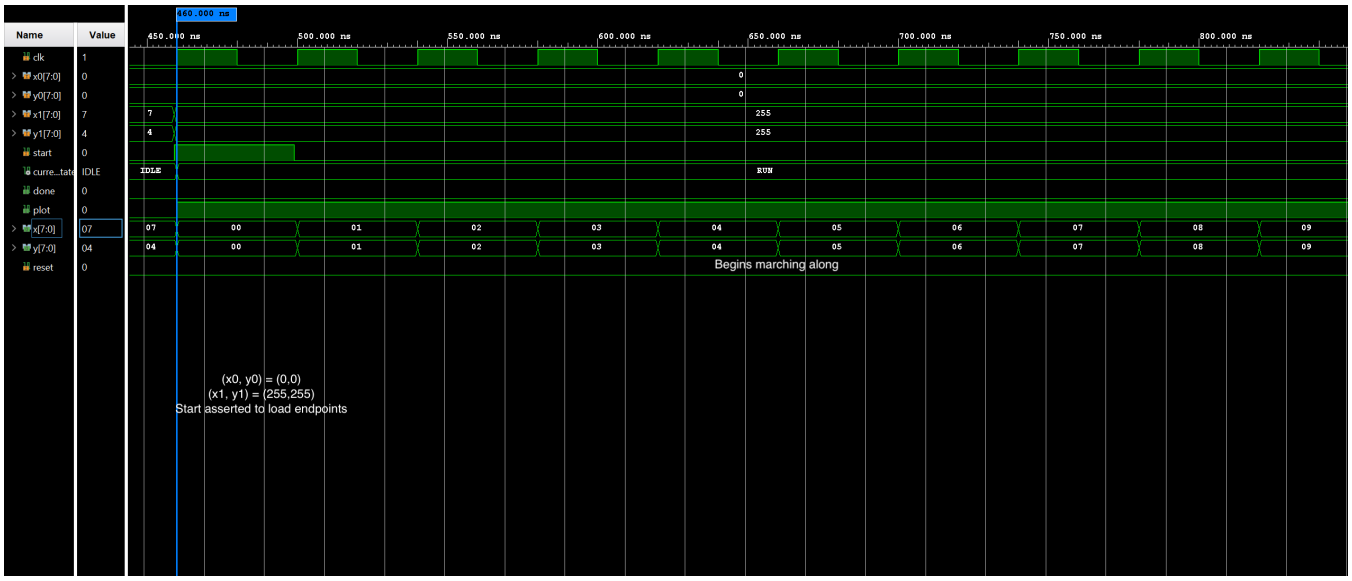

Figure 33: Bresenham Simulation (Image 2)

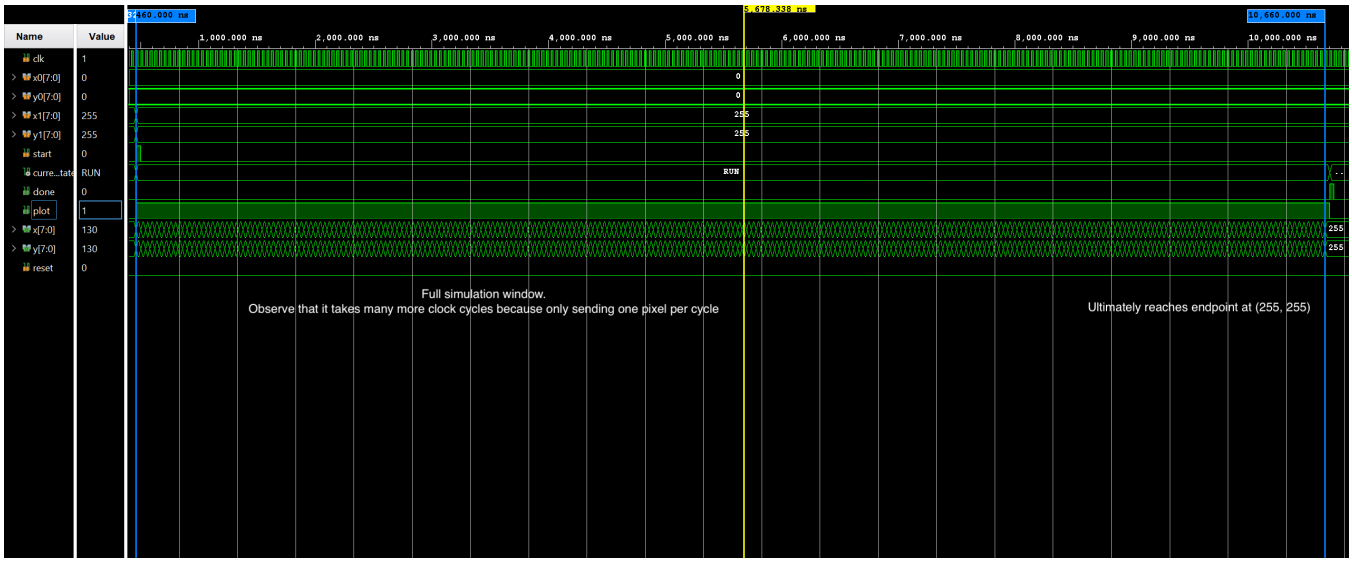

Figure 34: Bresenham Simulation (Image 3)

3.3 Bresenham_Receiver (bresenham_receiver.vhd)

Because we knew the Bresenham module was working, we only had to ensure that the receiver FSM was correctly applying the Bresenham algorithm between every combination of 4 points (6 edges). We simulated the behavior of the math manager and framebuffer by stimulating the vertices, new_vertices, and clear_fulfilled signals. Note that the siulation was done with a 100 MHz clock before we decided to use a 25 MHz clock, but the functionality of the receiver is not affected by the clock speed. We did not do a hardware validation of this module, instead opting to test it as part of a subsystem. For the screenshots, see sec. 3.13.

3.3.1 Behavioral Simulations

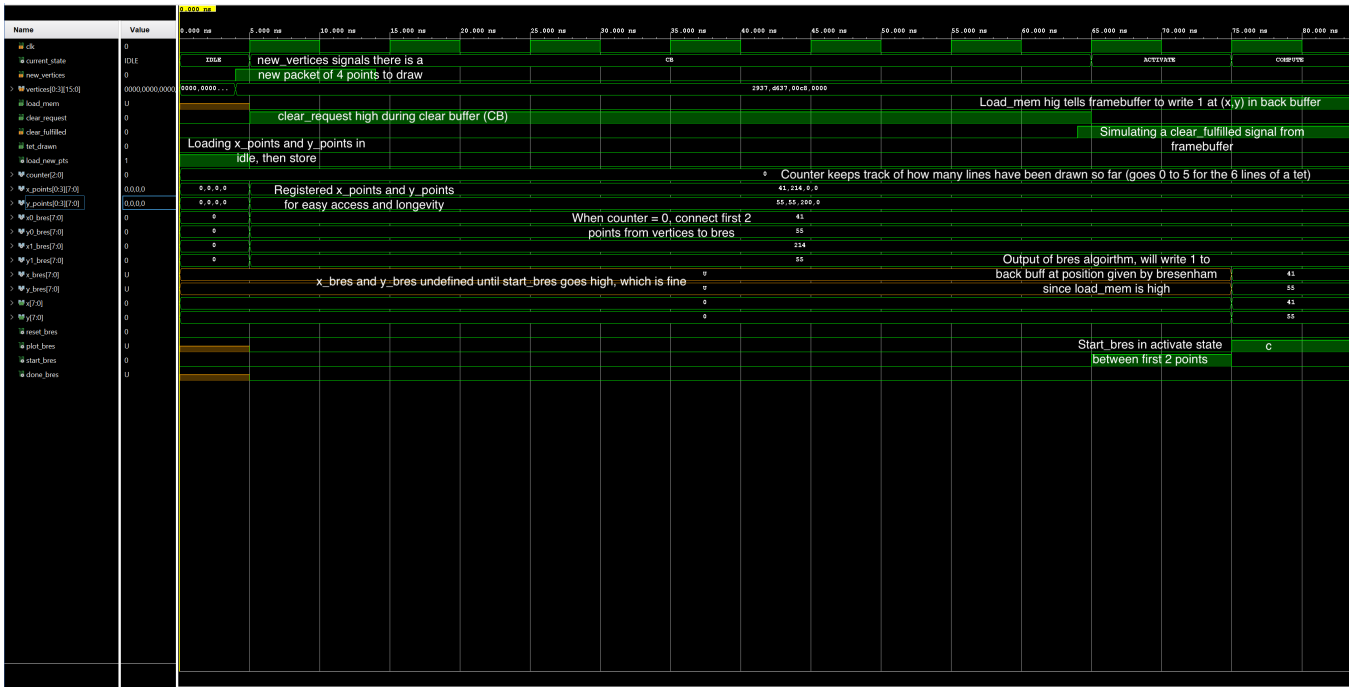

Figure 35: Bresenham Receiver Simulation (Image 1)

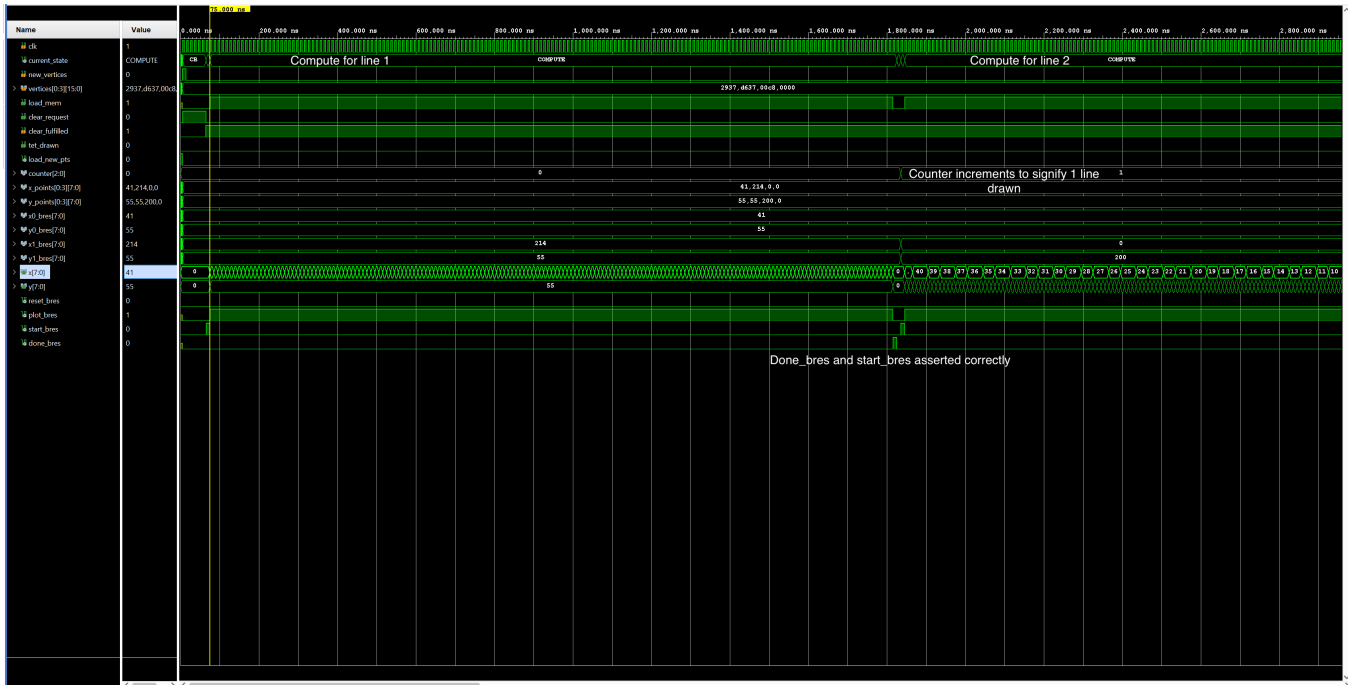

Figure 36: Bresenham Receiver Simulation (Image 2)

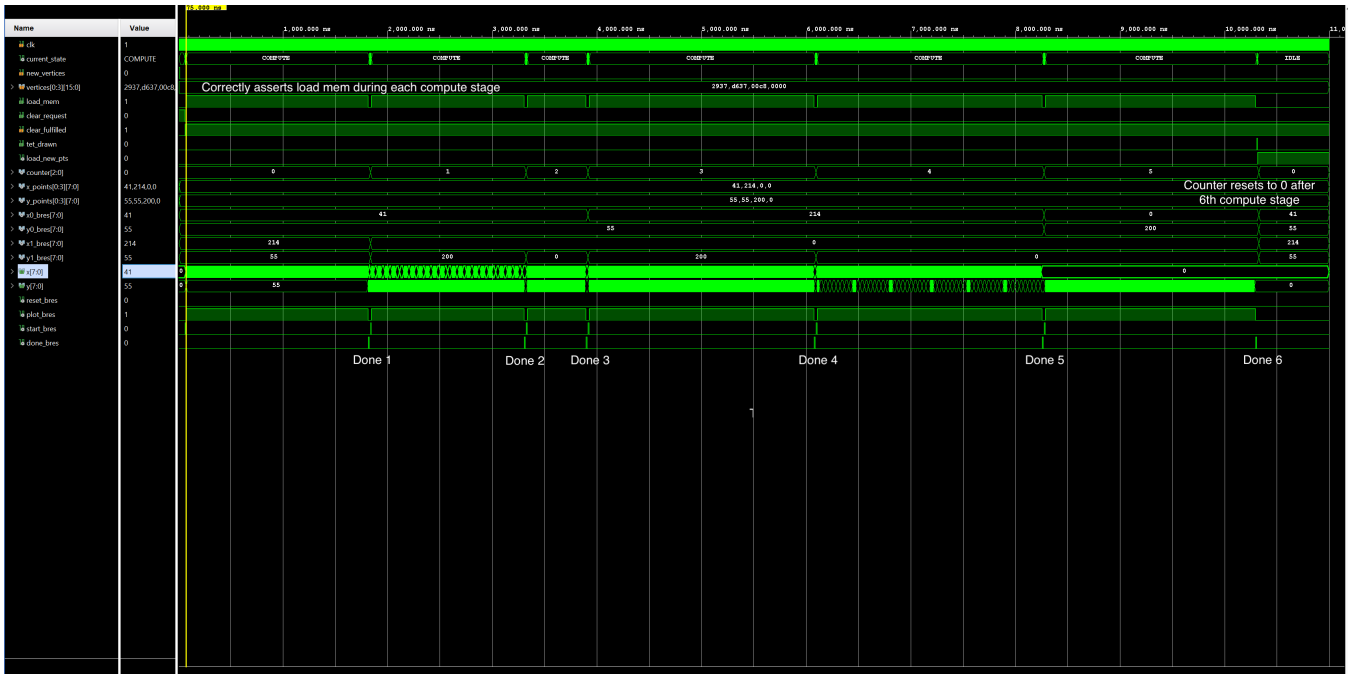

Figure 37: Bresenham Receiver Simulation (Image 3)

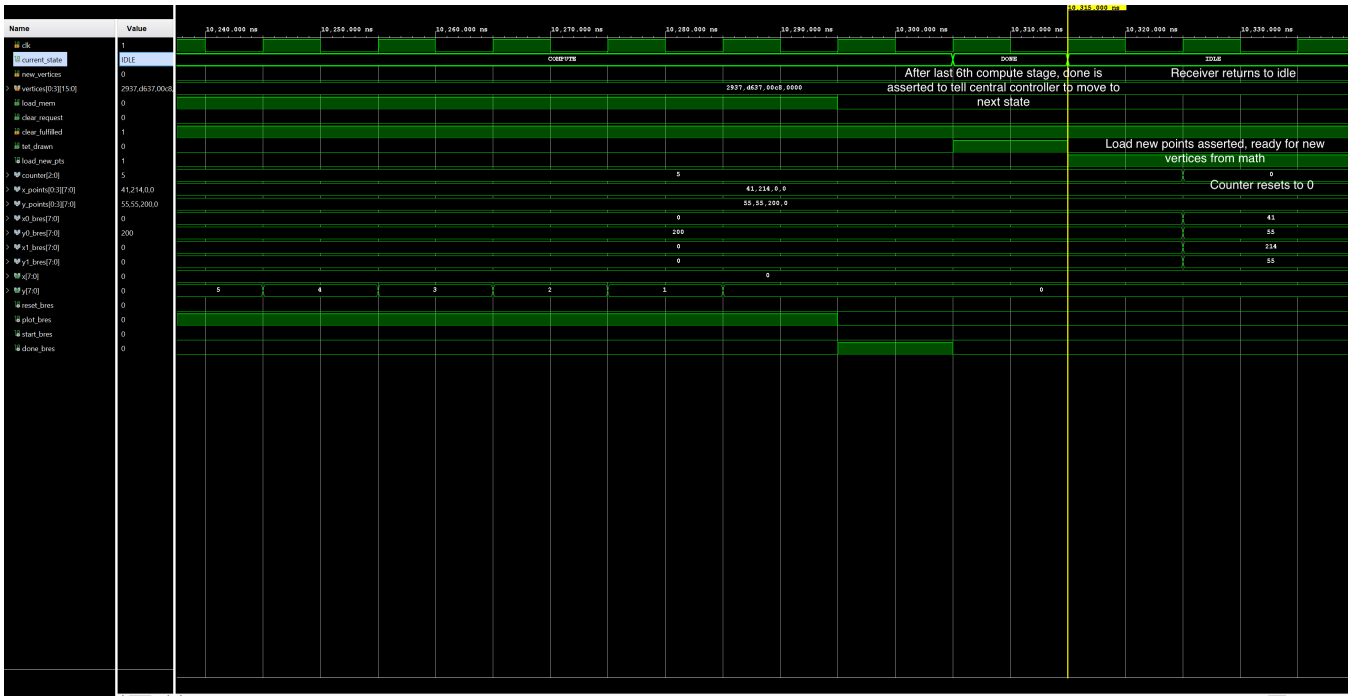

Figure 38: Bresenham Receiver Simulation (Image 4)

3.4 Framebuffer (framebuffer_v2.vhd)

I decided to simulate and validate the framebuffer module as a component of the graphics manager subsystem rather than as a standalone module because it relies heavily on the bresenham_receiver and vga_controller modules to provide read and write addresses, which would be tedious to stimulate in simulation. For the screenshots, see sec 3.123

3.5 Graphics_Manager (graphics_manager.vhd)

Similarly, we tested the graphics_manager subsystem, which encapsulates and connects the vga_controller, bresenham_receiver, and framebuffer modules using a shell called graphics_man_test_shell. For the screenshots, see sec. 3.13.

3.6 Parallel_Math (Math_Manager.vhd)

NB: All modules used in defining the Math Manager did not require indpendant hardware validation, but were instead verified using subsystems.

To test and validate, we wanted to see points rotate correctly and then confirm if the output points visually look correct in terms of rotation and then the correct shape of the tetrahedron. Therefore our simulation applies 2 sets of rotations and projection. We took the outputted point packets, graphed them on a 2D graph and visually confirmed that they both rotated in the proper direction and that the projection preserved the general shape of the tetrahedron how we need it to. Since we knew that all modules utilized in the Parallel_Math we felt it was redundant to confirm if the rotation was correct (Since rotation was confirmed in its own separate testbench).

3.6.1 Behavioral Simulations

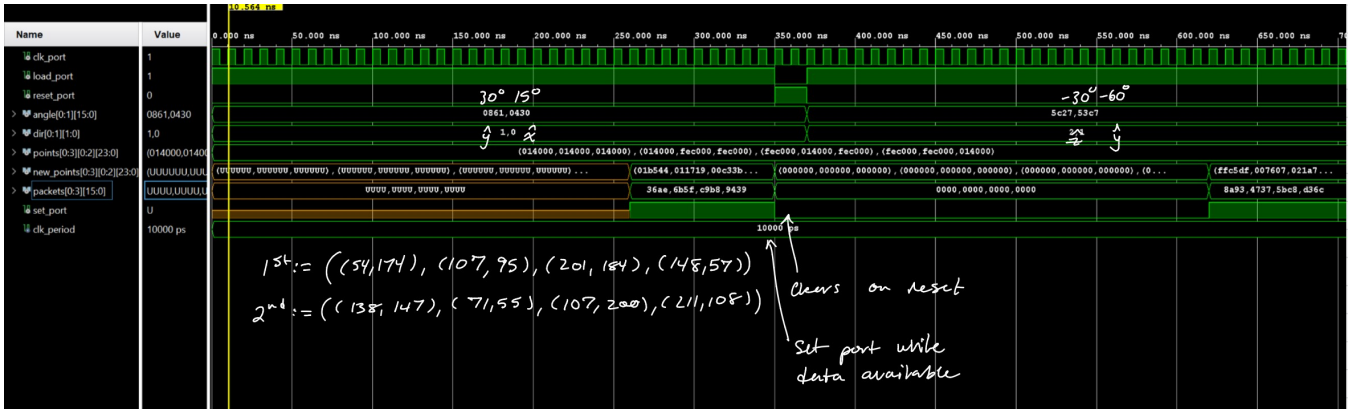

Figure 39: Math manager annotated simulation

To confirm we manually plotted points on a 256x256 plot (Same as framebuffer). The two plots below are the first and second rotation to projections. Notice in both we have rotated in the expected axis and then clearly the projection forms the tetrahedron. In practice we wouldn't rotate by such large angles.

3.7 Update_Point (Linalg_Update.vhd

We perform the testbench for this module in the Parallel_Math module since it's essentially a wrapper over four of the Update_Point entities. See section 3.6 for more information about our testing methodology.

3.7.1 Behavioral Simulations

See section 3.6 (Parallel_Math) above for simulation.

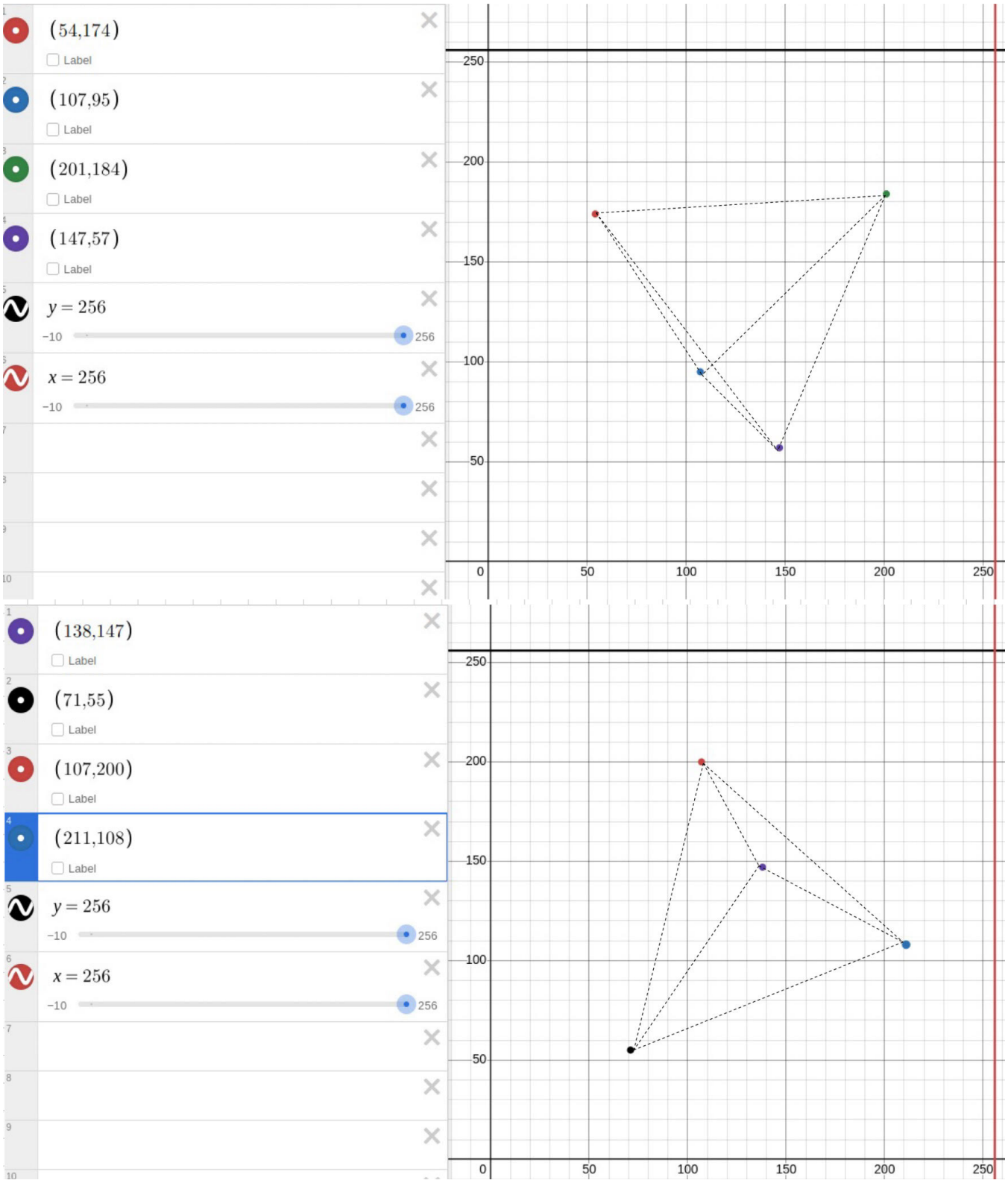

Figure 40: Validation of Math Manager and Update_Point showing two examples of tetrahedrons post rotation and projection.

3.8 Rotation Matrix Multiplication (rotation.vhd)

Our testing methodology for this component was simple. Ensure that the output points for a rotation approximate the expected points for the given angle. Because the points are fixed point representations, there will be error, however we want to ensure that the sign of the output is correct and the value is close to the integer. To validate, we compare the output to expected value and show that the reset and load ports work as intended. We do not explicitly test the subcomponent accumulate_rotation since it's validity is directly reflected in the output of the overall module.

3.8.1 Behavioral Simulations

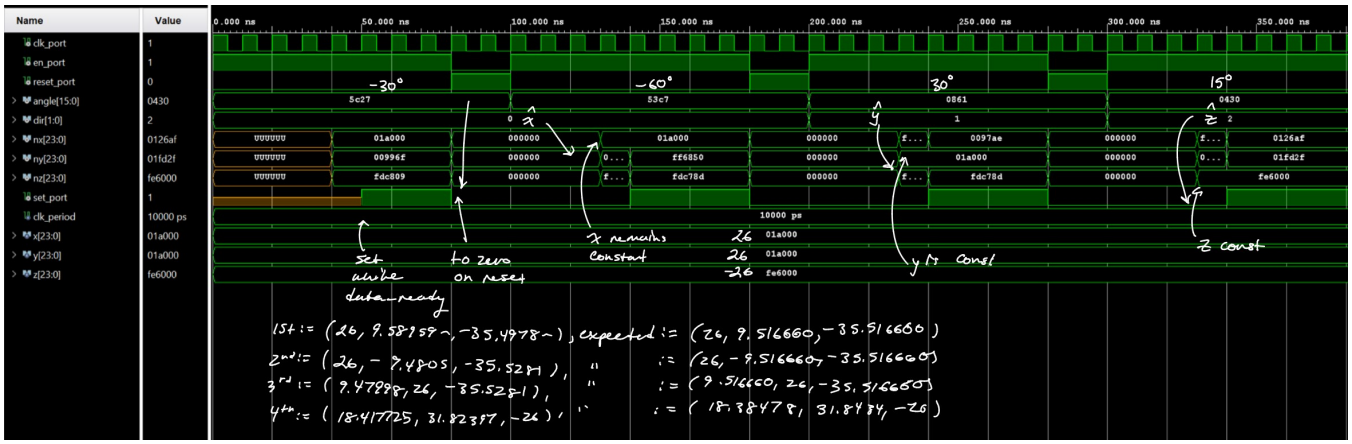

Figure 41: Rotation Behavioral Simulation.

It might be hard to see in the image but the key takeaway is that our rotation matrix multiplication computes the correct rotated points with relatively low error.

3.8.2 Subcomponent A: Sine Lookup Table

We request the values of $\sin\theta$ and $\cos\theta$ for $\theta = \pi/2, \pi/3$ to show that the output angle is correct and that the cos_en port correctly fetches the value of cosine. Our waveform confirms that behavior.

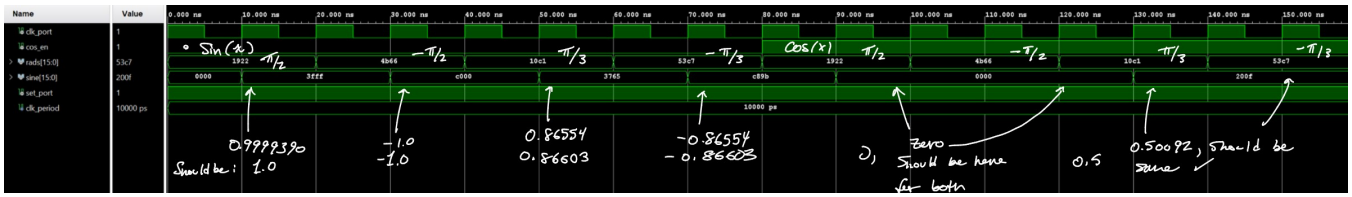

Figure 42: Sine Lookup Table Simulation.

3.8.3 Subcomponent B: Set Operands

We wanted to test and ensure that the values registered in set_operands reflect the correct operands for each axis. Additionally we want to ensure set_port asserts when operands are available.

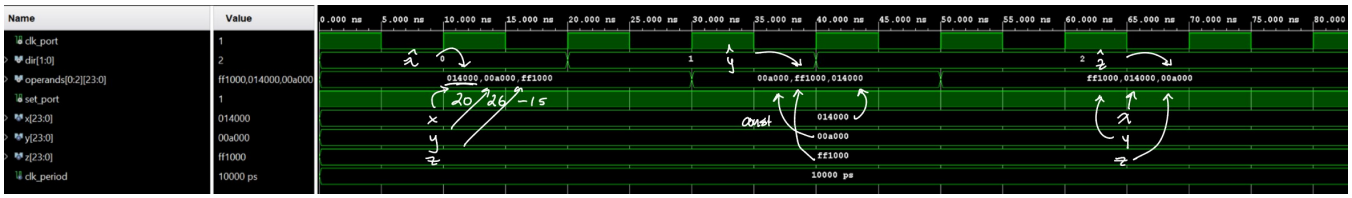

Figure 43: Set Operands Simulation.

3.9 Projection_Matrix_Multiplication (projection.vhd)

We wanted to confirm for our initial points that the projection generated visually matches what we expect to see for the starting projection of our tetrahedron. Our test plan therefore was to ensure that for some initial points our matrix multiplication is able to properly output the correct 2D points on the viewport. We used the module to compute points then manually plotted them to confirm their validity.

3.9.1 Behavioral Simulations

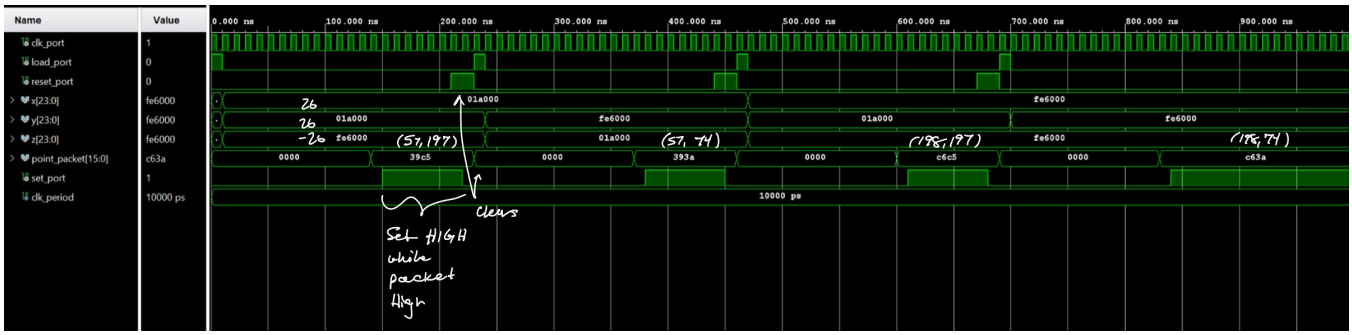

Figure 44: Projection Behavioral Simulation

Most of the annotations involve noted certain behaviors we want to be constant in all math modules such as the reset port clearing output registers, or that nothing occurs until load_port is asserted. From the four computed projected points we made the following graph and annotations to show that the module's output is in fact valid.

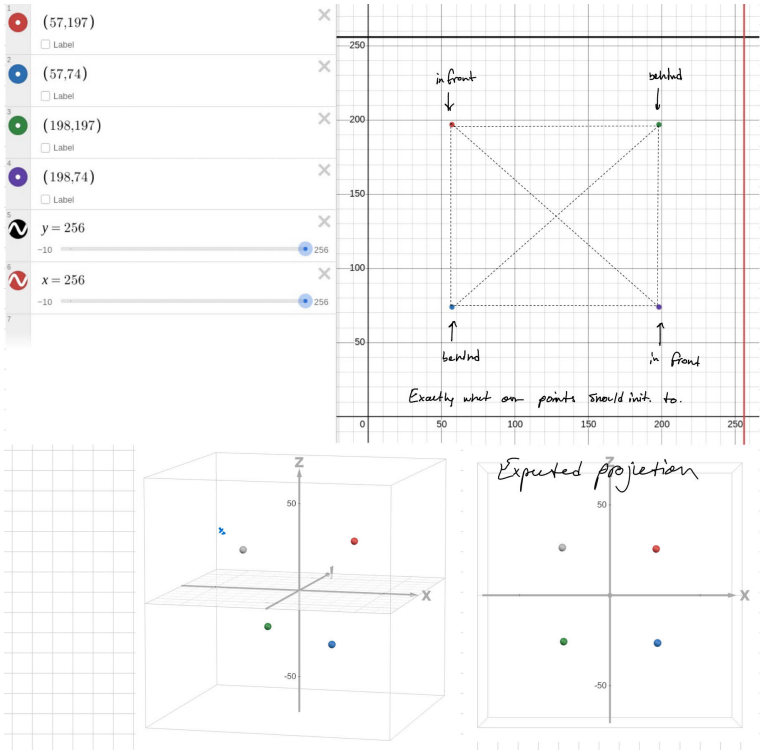

Figure 45: Validation of Projection Module

3.10 Reciprocal (reciprocal.vhd)

The testbench should show that we can retrieve the reciprocal of a value that is relatively close from what we would expect the reciprocal to be. Several limiting factors restrict how close our estimation can be. We use 11.12 notation for the value which limits the amount of bits allocated to the fractional and integer part. We can only perform two iterations of newton's method which means we truncate early and it is possible our initial guess was quite far from the real value. Finally, our lookup table is fairly low resolution and only for our normalized value. Hence when we de-normalize we incur some loss of fractional bits which removes precision further. Therefore for our validation we hope to see a value that is relatively close. Furthermore, since we clip values of $z < 8$, we will not need to retrieve a reciprocal for a value less than 8 in practice. Our algorithm becomes more accurate for larger z, hence we minimize a lot of the error that might be incurred from poor estimations.

3.10.1 Behavioral Simulations

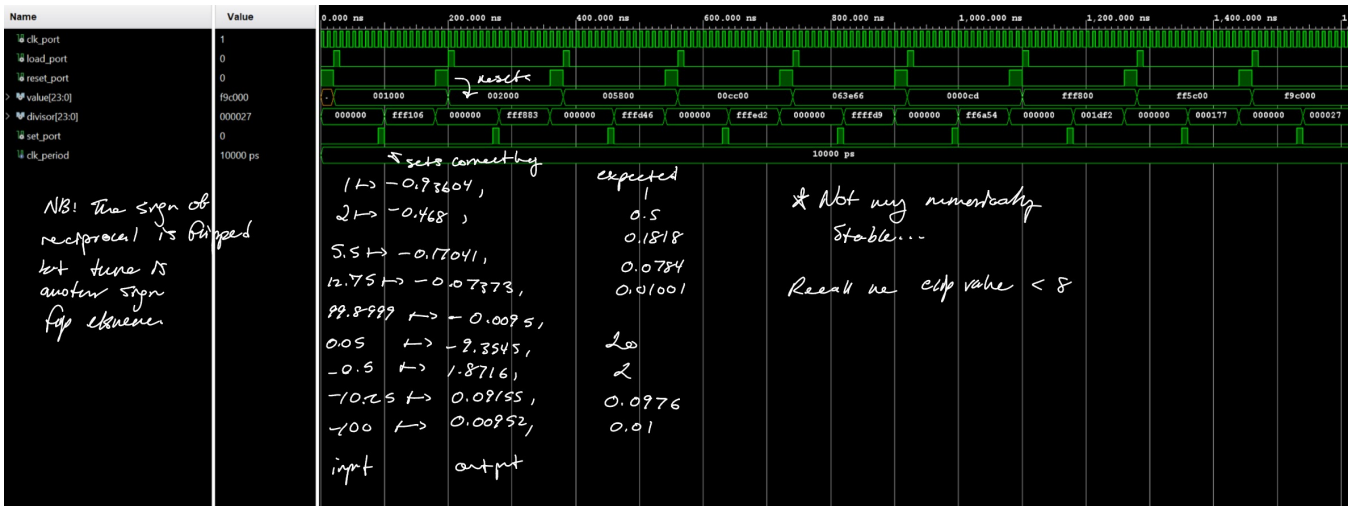

Figure 46: Validation of Reciprocal function

It might be hard to see but the key takeway should be that for low z the estimation is unfortunately not great; however as z increases, like at 1/5.5, the estimation begins to converge much better which means our division will be much more accurate in practice with our clip value of 8. We also want to note that the output of the reciprocal table, the simulations here did not include logic that "remembered" the original sign of the value and reapplied it at the end.

3.10.2 Subcomponents A and B: Newton's Method (With validation of Newton LUT)

We omitted the testbench of Newton's method for two reasons. First, the output of the newton's method entity is agnostic to the original normalization factor that our value was divided by. Secondly, our newton's method output flips the sign of the value, something that is addressed by the caller of the parent module reciprocal. Therefore we determined that testing should be done at the level of the reciprocal entity where we determined if the newton's method was at fault if the mantissa and lut response are confirmed to be correct.

3.11 Angle_Dir_LUT (angle_dir_lut.vhd)

The Angle_Dir_LUT is tested as part of the overall system at the top level in Central_Controller. Because it such a simple component, we decided to test it alongside the reset of the system. See sec. 3.14

3.12 UART_Receiver (uart_receiver.vhd)

First, we tested the receiver in simulation and made sure that the FSM was behaving as expected. We quickly resolved minor bugs such as missing sensitivity lists on the FSM during this stage. We felt that hardware validation was important for this component because it is inefficient to simulate sending many characters in a row. Therefore, we created a shell that prints the data read by the UART receiver to the 7 segment display in hex. This allowed

me to easily verify accuracy and behavior such as held and rapidly changed keys. For a while, we were stuck with inaccurate values in hardware even though my simulation looked good. Eventually, we realized that we had only simulated one byte of data, and when we sent a second byte, it was not behaving as expected. Validation of this module was a success when the receiver worked accurately and responsively in hardware.

3.12.1 Behavioral Simulations

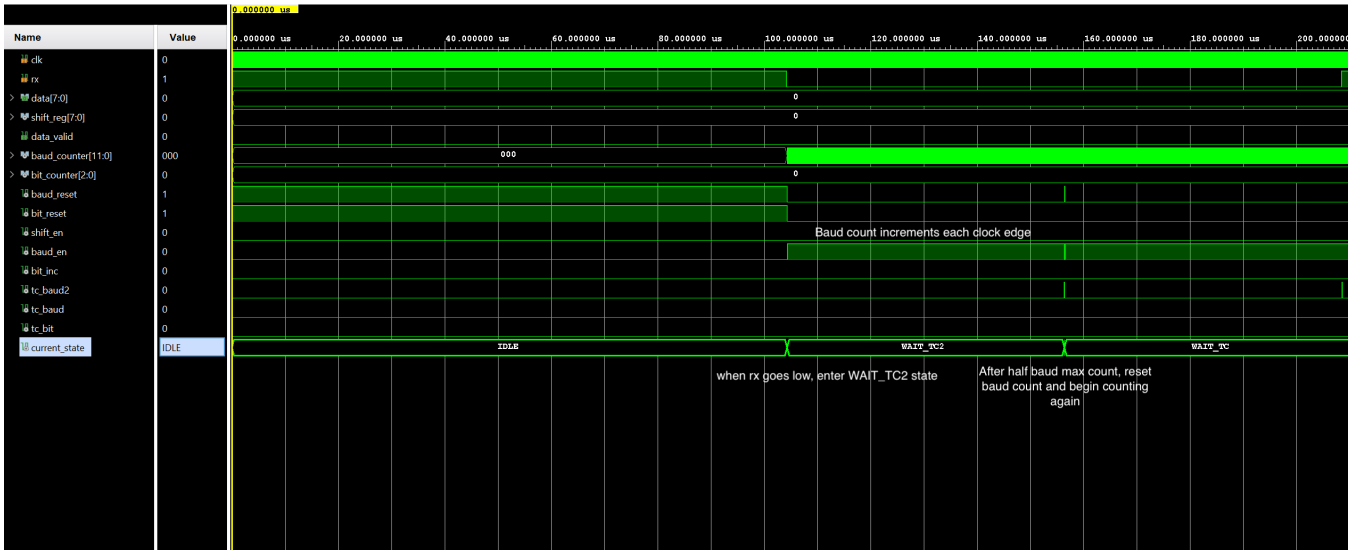

Figure 47: UART Receiver Simulation (Image 1)

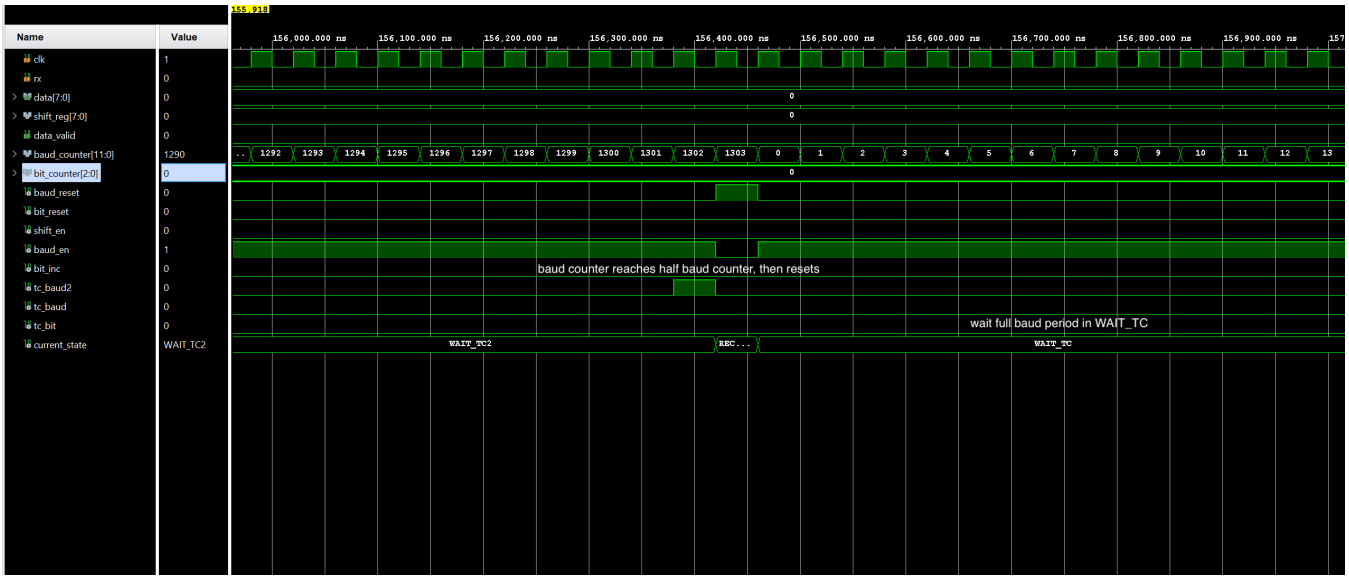

Figure 48: UART Receiver Simulation (Image 2)

Figure 49: UART Receiver Simulation (Image 3)

3.12.2 Hardware Validation

Here is a link to a video of the hardware in action: HERE. For convenience, below is an image of the setup when pressing the A key (ASCII 61):

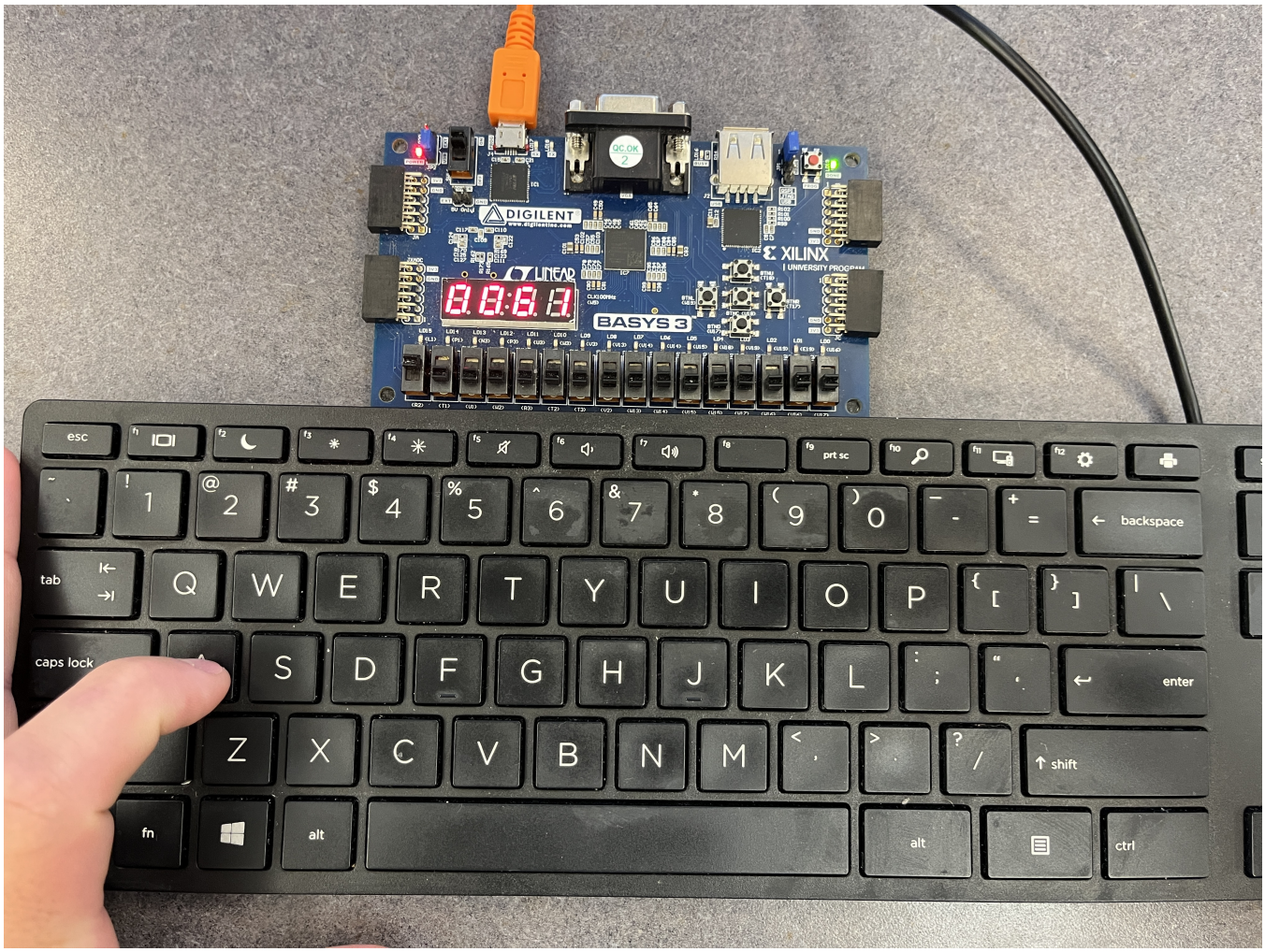

Figure 50: UART Receiver Hardware Validation

3.13 Subsystem: Graphics_Man_Test_Shell (graphics_man_test_shell.vhd)

I used the graphics_man_test_shell to simulate the functionality of the Bresenham_receiver and framebuffer modules simultaneously because they interact closely. In the shell, we connected the these two modules along with the vga_controller. Furthermore, in the shell, we created a process that artificially asserted new_vertices and assigned one of two packets. It then counts to 12,500,000 clock cycles before reasserting new_vertices with the other packet (a new shape). This results in two frames being drawn, alternating at 1Hz. This is a good test because it shows that the buffers swap correctly and that This artificial stimulation was just like a testbench, but allows/ for synthesis and hardware validation. For the simulation, we simply provided the graphics_man_test shell with a 25 MHz clock and observed the states and signals of the Bresenham_receiver and framebuffer modules. We considered this test as "sucessful" when we could see that the Bresenham_receiver and framebuffer were communicating correctly, as the screenshots describe.

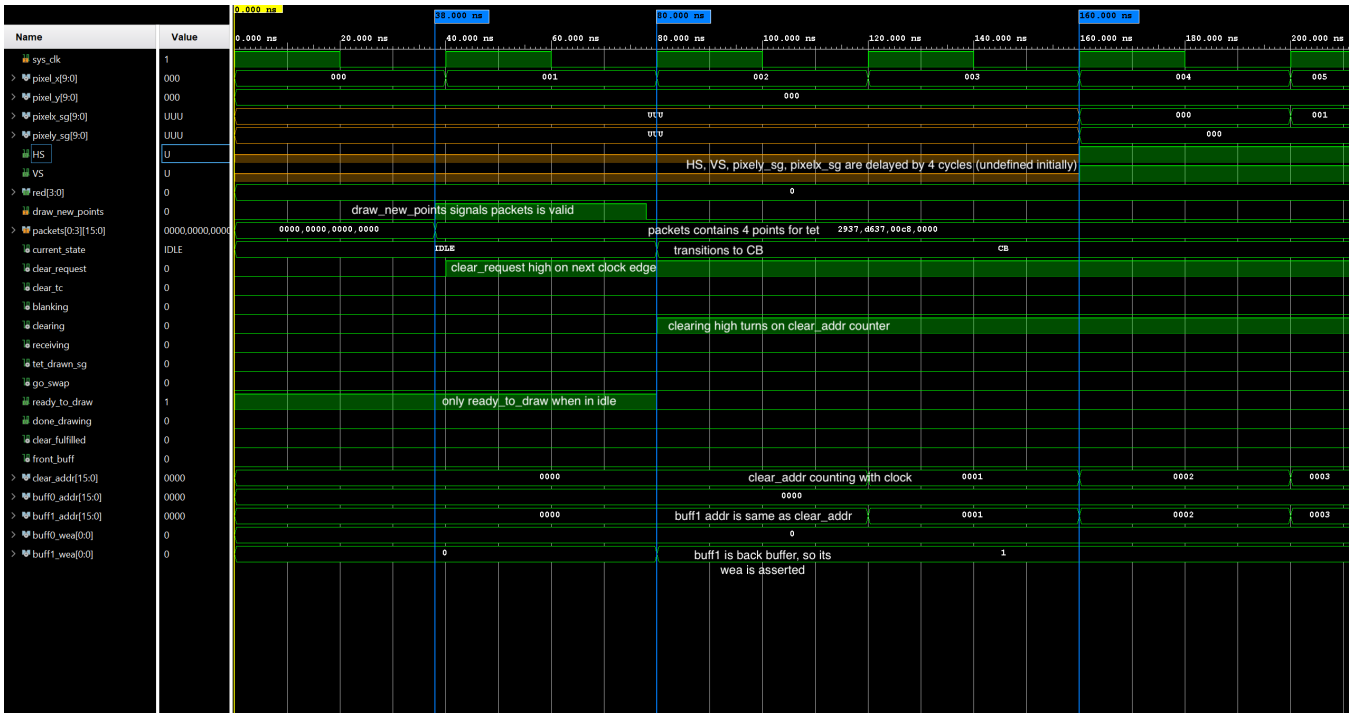

Figure 51: Graphics Man. Test Shell Simulation (Image 1)

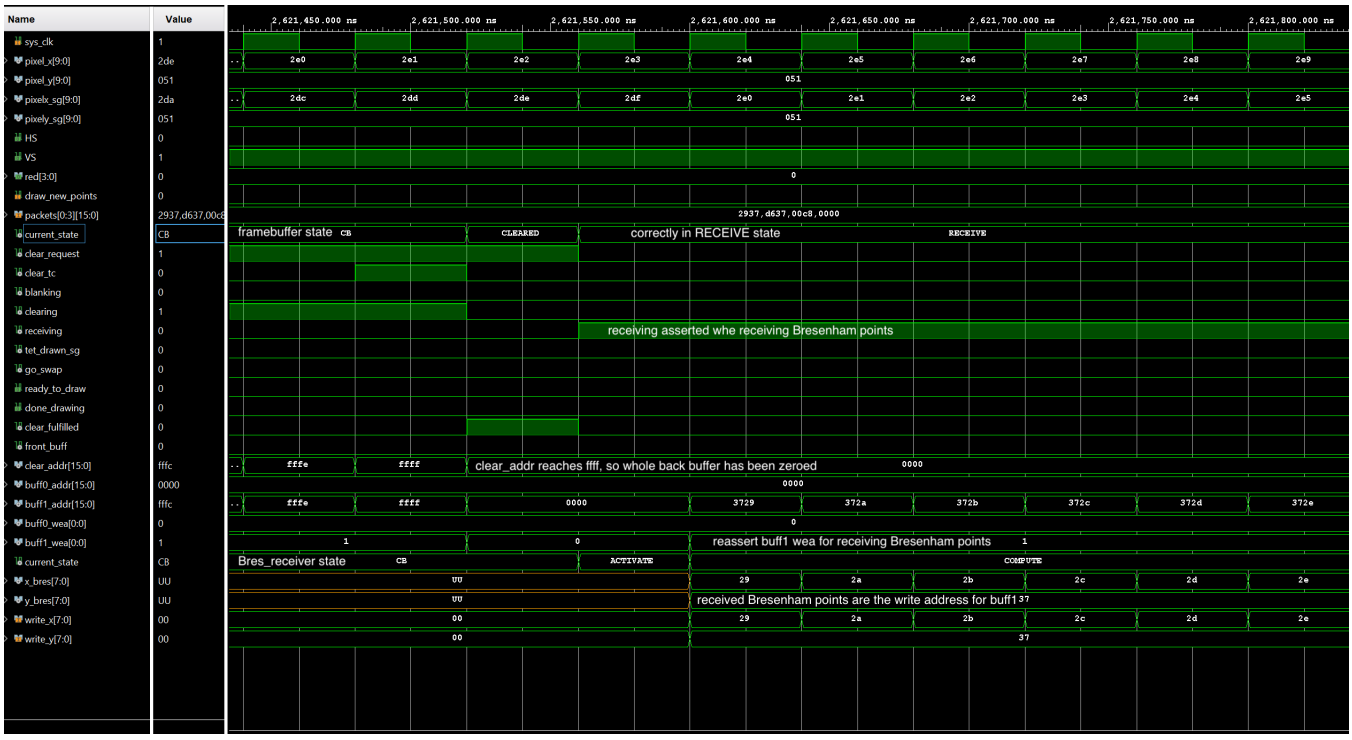

Figure 52: Graphics Man. Test Shell Simulationn (Image 2)

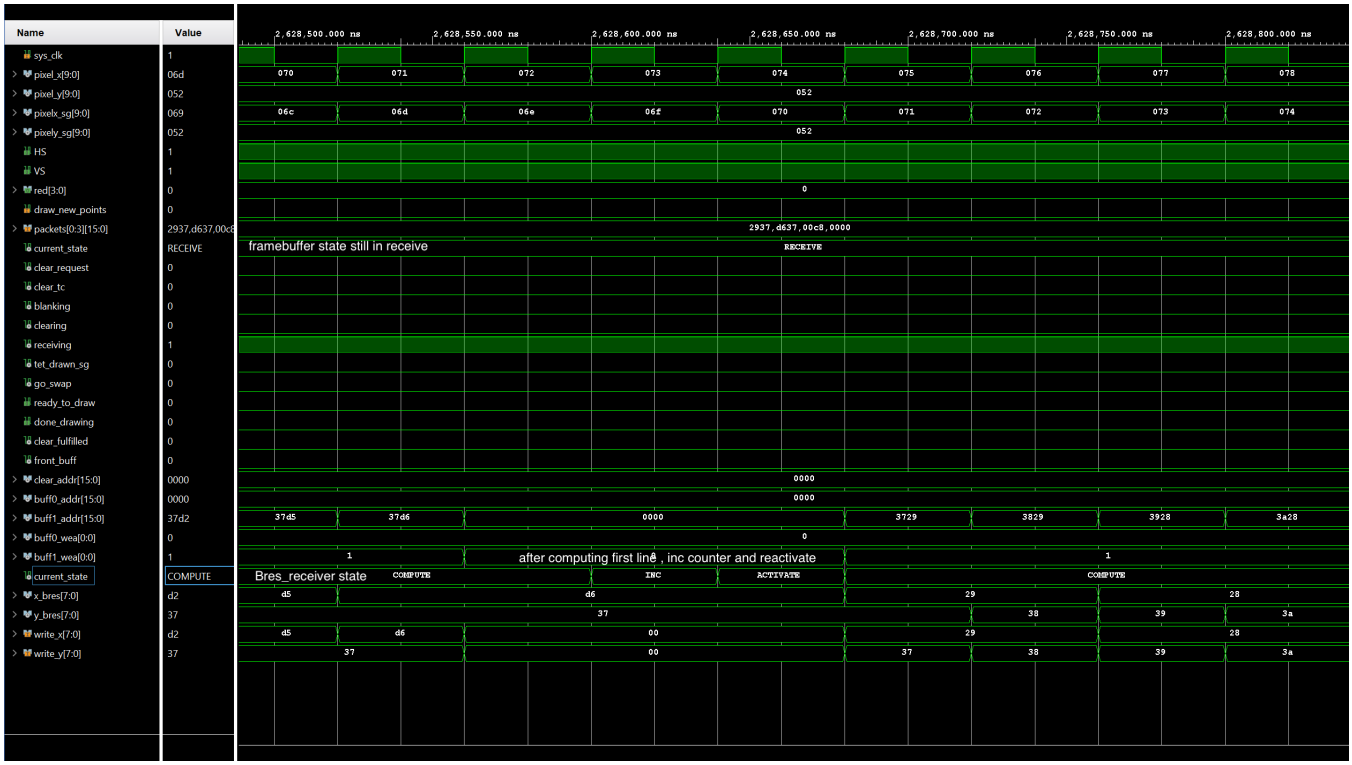

Figure 53: Graphics Man. Test Shell Simulation (Image 3)

Figure 54: Graphics Man. Test Shell Simulation (Image 4)

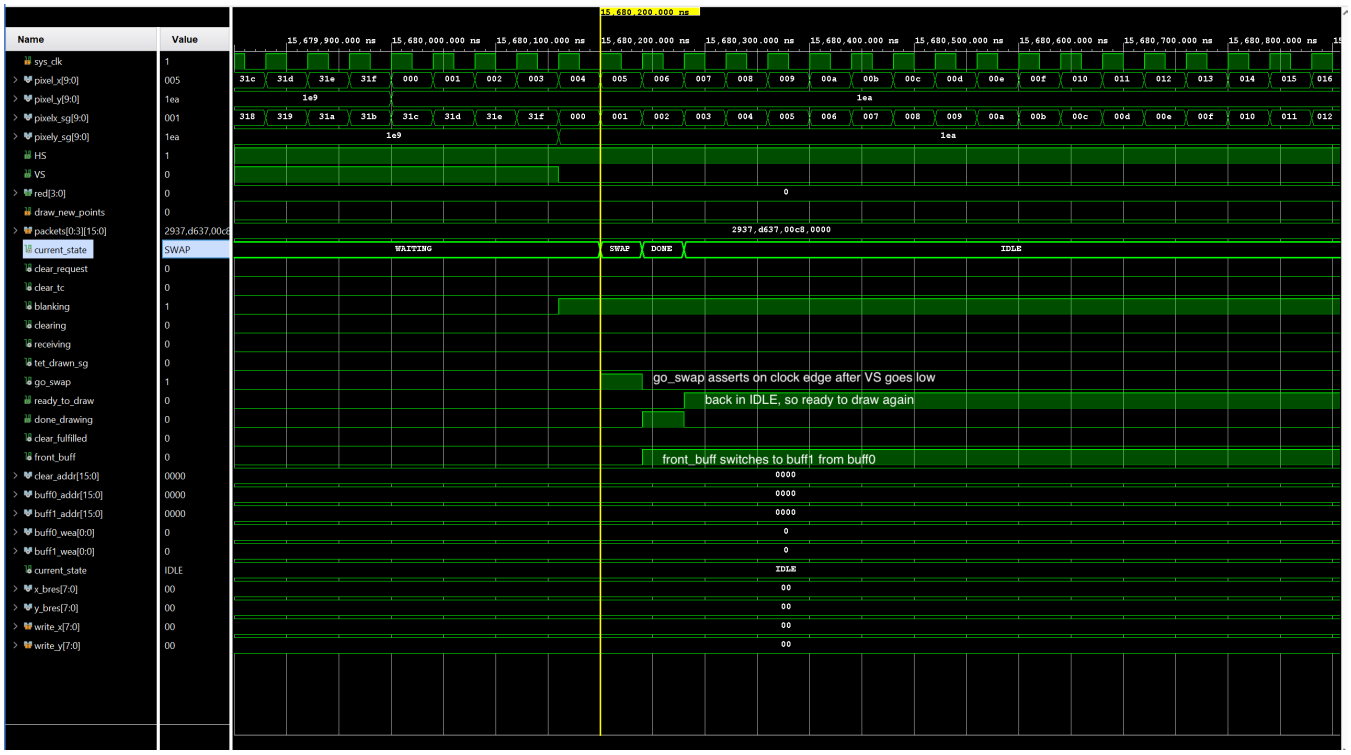

Figure 55: Graphics Man. Test Shell Simulation (Image 5)

Figure 56: Graphics Man. Test Shell Simulation (Image 6)

3.13.1 Behavioral Simulations

3.13.2 Hardware Validation

Here is a link to a video of the test shell in hardware: HERE. For convenience, below are photos of the two frames on the monitor.

Figure 57: Graphics Man. Test Shell Hardware (Image 1)

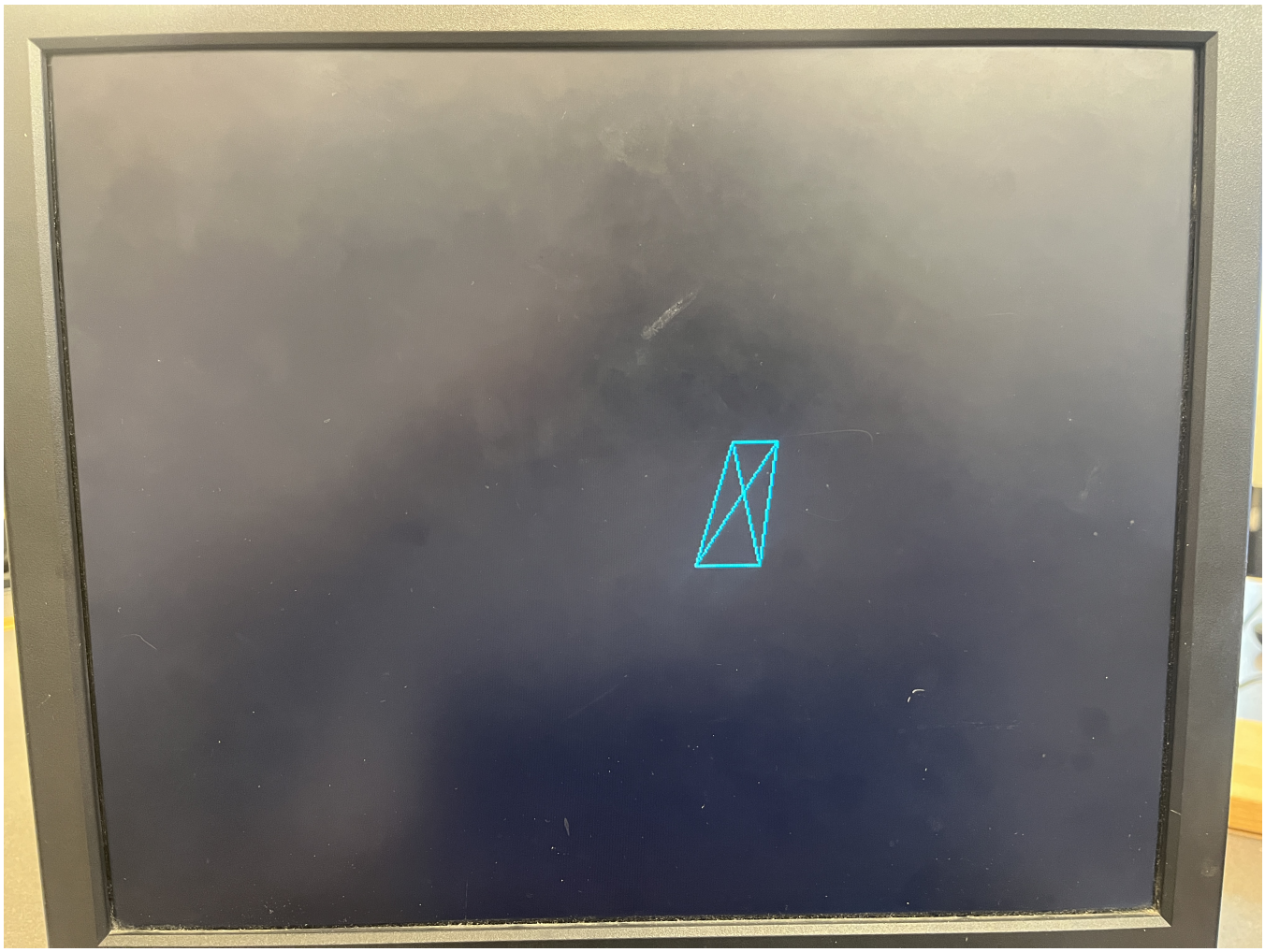

Figure 58: Graphics Man. Test Shell Hardware (Image 2)

3.14 Overall Design: Central_Controller (top_level_controller.vhd)

To simulate the overall design, we focused on ensuring that the top_level_controller FSM was working correctly. Using a testbench, we simulated the initialization, showing that the FSM was loading and drawing the default tetrahedron at the start. We the waited for a period and simulated pressing the W key by sending its ASCII key using UART to the RsRx_ext_port input. We then observed the correct mapping of this key to an address and direction, proving that the Angle_Dir_LUT module was working correctly. We also observed correct behavior for the math computation by the math manager and drawing to the screen, coordinated by the central_controller FSM. Because of the difficulty of debugging VGA through a waveform simulation, we decided to debug the design further in hardware.

The first major issue we ran into was the tetrahedron jumping wildly due to a vertex overflowing off the view port and wrapping around. Because the projection module divides by the z coordinate to incorporate perspective, when projecting points with z coordinates close to 0, the tetrahedron would rapidly grow in size towards infinity. This was solved by using the common graphics technique of clipping the z divide. If the z coordinate is smaller than some threshold, the projection module divides by that threshold rather than the true z value, preventing the projected point from blowing up. Through experimentation, we found that a clipping threshold of 8 provided the correct amount of perspective while avoiding clipping. Changing the size of the tetrahedron affects the optimal clipping threshold to use, so we experimented with both parameters, making the tetrahedron as large as possible on the monitor.

Another issue was that the first press of a new key would repeat the last operation before performing the new key's operation correctly. This led to the addition of the WAIT_MATH state to the central FSM, allowing the new angle and dir to propagate down into all math modules before activating. This addition solved the input issue.

Hardware validation also allowed us to tune our code to facilitate natural rotation. For example, originally, the W and S keys (up/down) rotated it along the z axis (out of the screen), which is not what a user expects. In the angle_dir_LUT for natural rotation, we optimized the keybindings. We also tuned the rotation angle per frame to provide a rotation rate that was smooth but responsive, settling on 1° per frame. At 60 Hz, it therefore takes 6 seconds to fully rotate. Finally, we chose to display the tetrahedron with a holograph-blue color that looked nice on the monitor. We considered hardware validation a success when it rotated as expected without clipping and had a good "feel" to use.

3.14.1 Behavioral Simulations

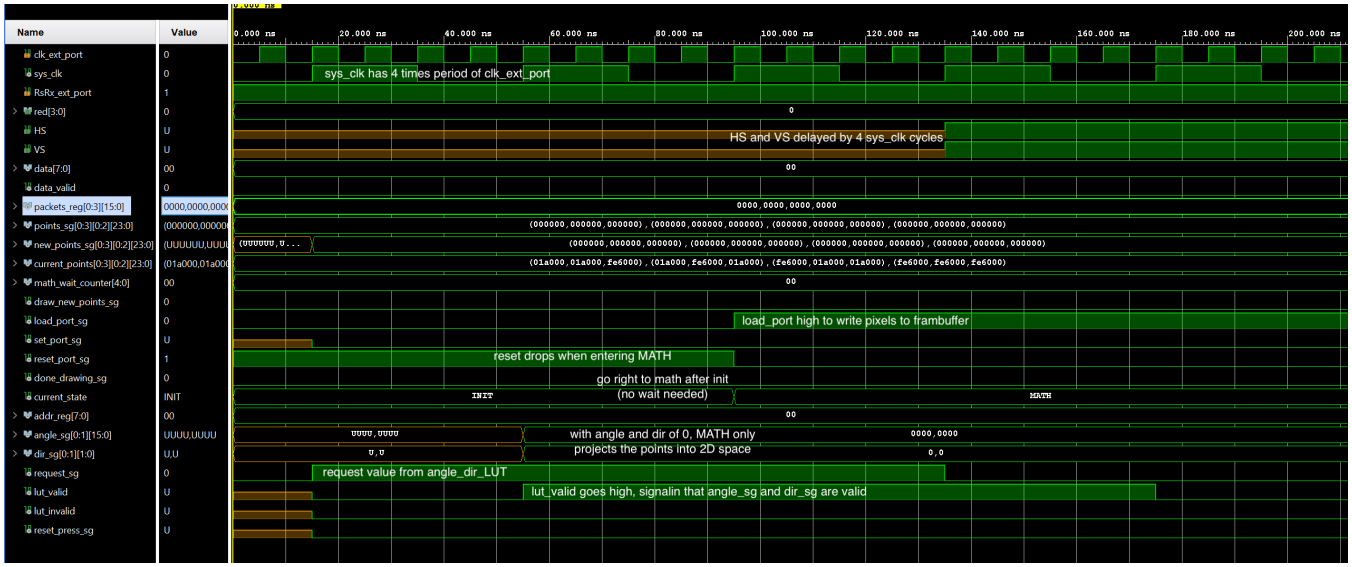

Figure 59: Overall Design Simulation (Image 1)

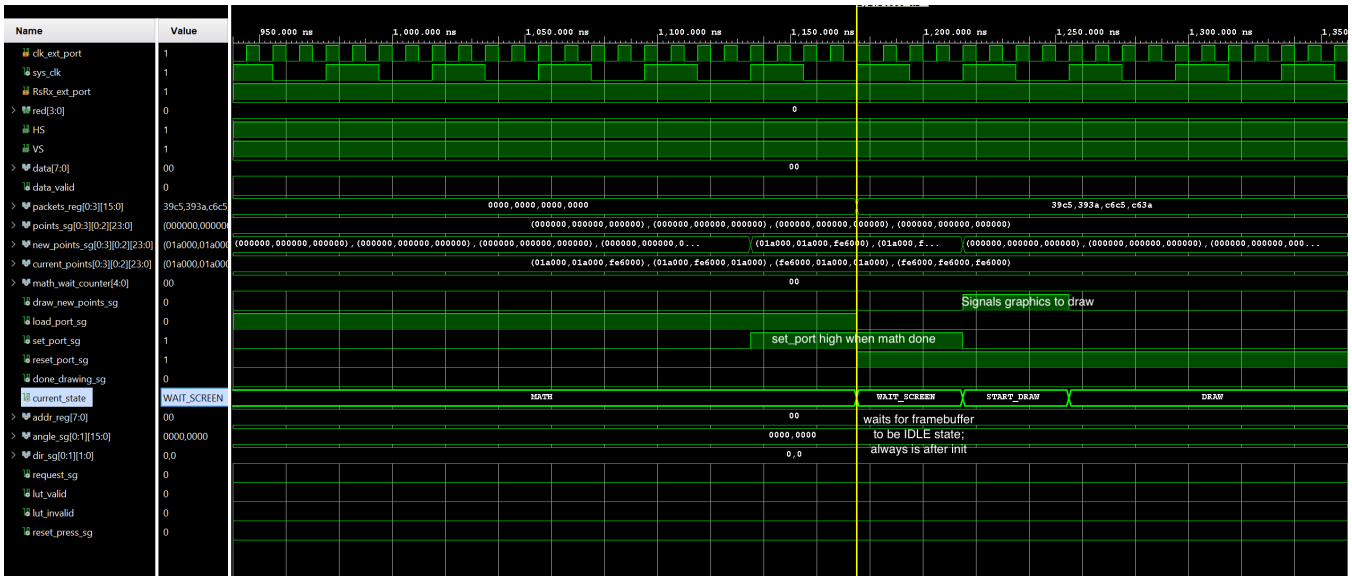

Figure 60: Overall Design Simulation (Image 2)

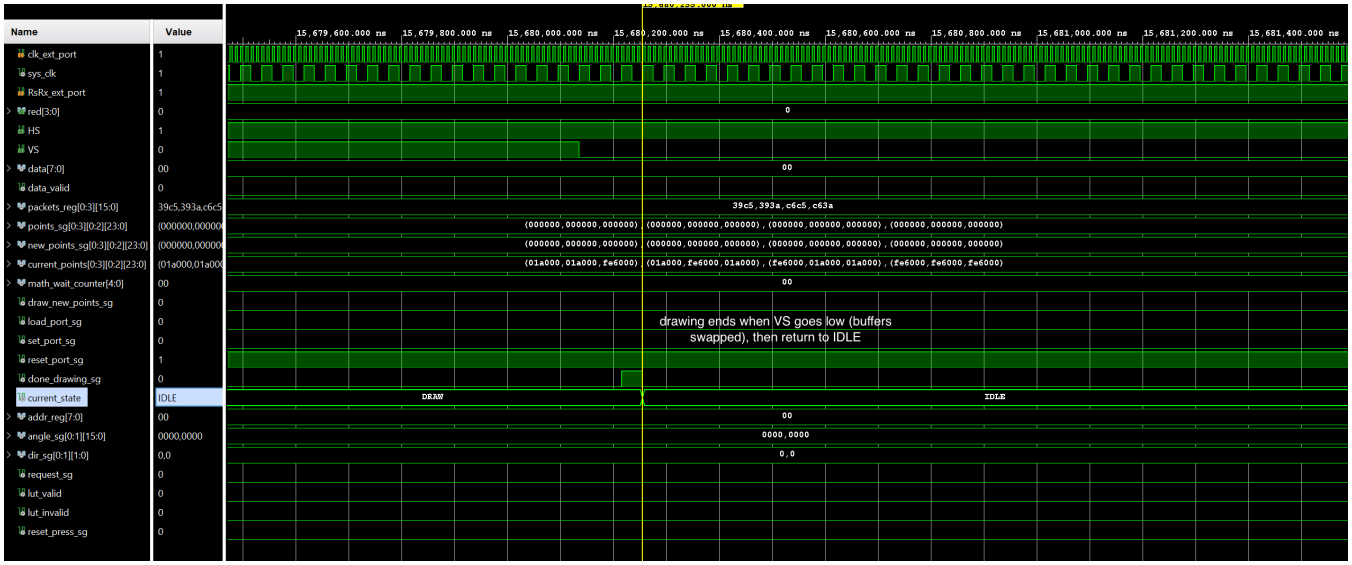

Figure 61: Overall Design Simulation (Image 3)

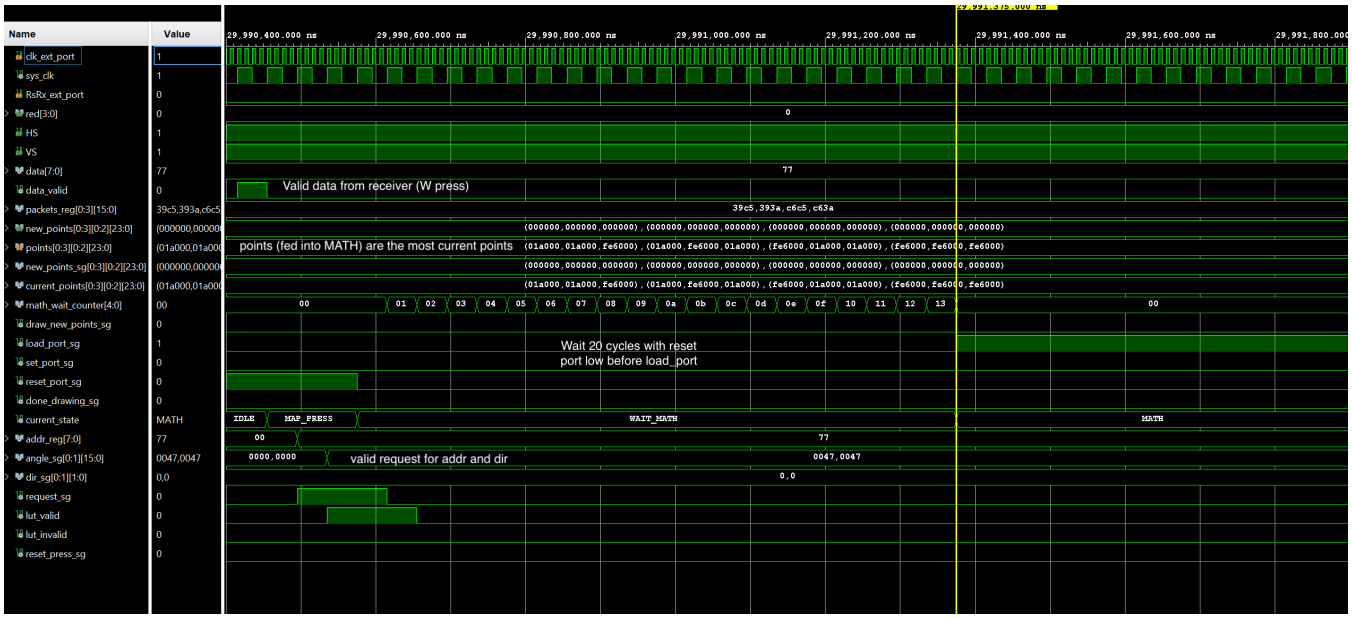

Figure 62: Overall Design Simulation (Image 4)

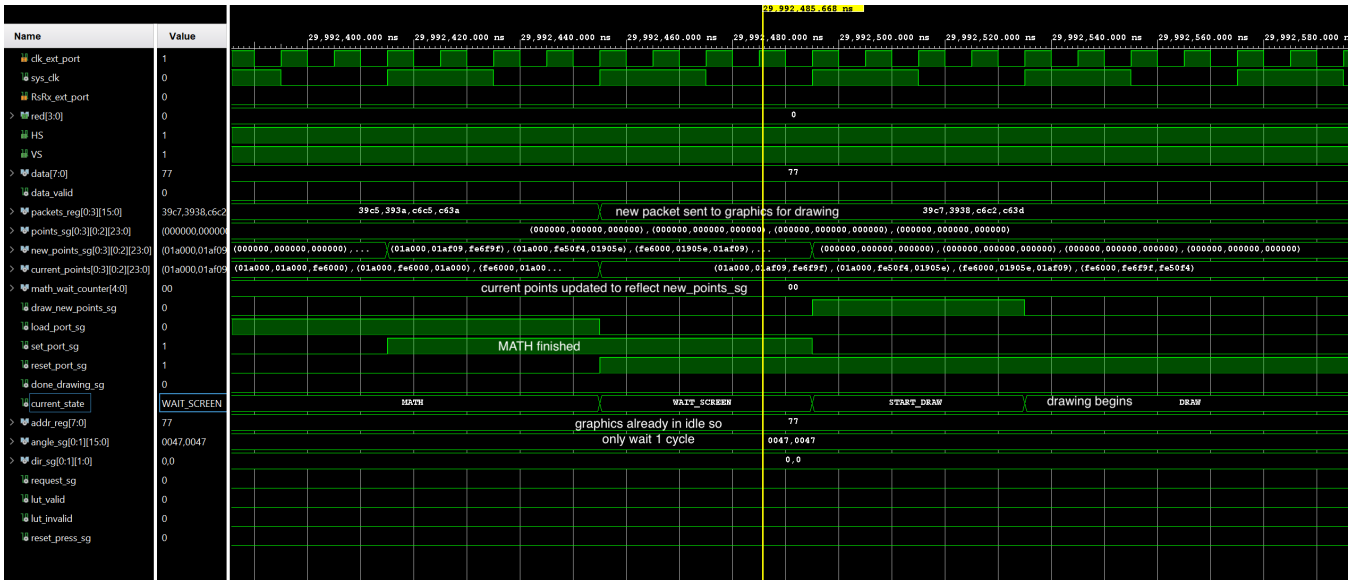

Figure 63: Overall Design Simulation (Image 5)

Figure 64: Overall Design Simulation (Image 6)

3.14.2 Hardware Validation

Here is a link to a video of the project in action: HERE

4 Analysis of the Design

4.1 Resource Utilization

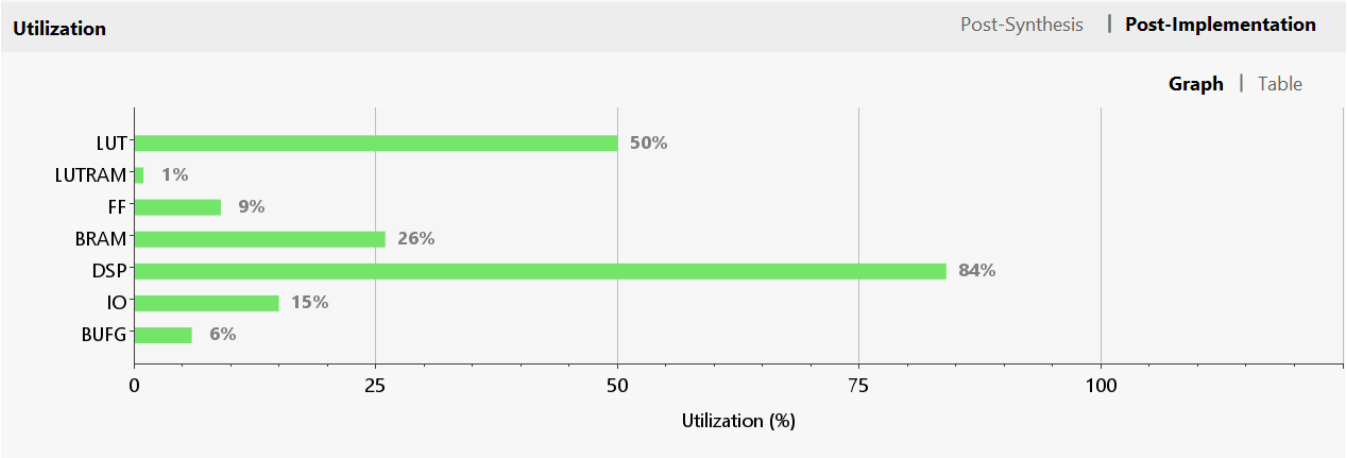

Figure 65: Utilization of FPGA Resources Tabularized

In Figure 65 we show the utilization percentages of the final implemented design. We utilized a majority of the DSP slices due to how we parallelized the math operations within the design. This value could be optimized significantly since the time between swapping framebuffers is so large relative to the time for one update_point module to run from start to completion. There are other optimizations that could be made to reduce this number. We use 24x24 bit multiplication in several places, but the closest multiplier the synthesizer can infer is 25x18 bit multiplication. Therefore it must allocate an extra DSP slice per multiplication in this form. We made this optimization in several places but for the sake of time opted to neglect doing so for all instances once we got a working version. All of our lookup tables were able to be inferred as block ram since they lacked reset ports which helps significantly with LUT usage, however we would've swapped to loading IP Cores with a .coe file had that been an issue. A large percentage of the LUT table usage comes from large datapaths in the math component, especially sections that could not be inferred as DSP slices. Furthermore the de-normalization logic in the reciprocal module utilizes runtime dependent shift logic, which from what we could tell, had a significant LUT cost. Spread out over 4 update_point modules, this incurred a significant LUT cost, and would be something we'd look to eliminate/mitigate given more time. The framebuffer was loaded as IP core BRAM which also contributes to us using about 1/4 the total BRAM. Finally, our FPGA utilized UART to communicate with PUTTY to get keyboard inputs that allowed us to rotate key press to rotating on specific axis.

4.2 Residual Warnings

I have selected the warnings that are relevant to the system functionality. Noteably, there are no inferred latches in our design. Synthesis

- "Unused sequential element was removed" in the Bresenham module
  - This is simply informational. The module works as expected.
- "Signal x_points is read in the process but is not in the sensitivity list" in the Bresenham_receiver
  - This is a process that updates based on current_state, and x_points only needs to be read when the state changes
- "3D RAM new_points_reg for this pattern/configuration is not supported. This will most likely be implemented in registers."
  - Self-explanatory
- "Resources of type DSP have been overutilized"

  - It will use regular LUTs for the remaining multiplys

- "Input pipelining: DSP input is not pipelined. Pipelining DSP48 input will improve performance."
  - This could be expanded on in future projects. There are warnings for output pipelining

4.3 Division of Labor

We knew (and warned) that our project would be a significant undertaking from the start. In order to streamline our design and allow for us to not waste any efforts, we decided to design our control top down to be two major managers. These managers would be black boxes to each other, allowing for us to write strongly decoupled modules that could be unified at the end to communicate to each other. We knew that we would need UART inputs as well as a top level unifying controller, however our initial plan was to work on our respective managers and as each person finished their manager, they would begin the auxiliary components needed to finish the implementation. Andrew worked on the Math Manager and the angle/dir lookup table to map keyboard inputs to rotations in specific direction. Ben worked on the Graphics Manager, the UART receiver, and the top level shell.

4.4 Future Work

Our project was less about rendering the tetrahedron and more about the process of building a graphics engine from scratch on the FPGA. Therefore, contingent upon some of the optimizations talked about in the utilization section, we have many directions we could take to improve this project even further. Adding more points is as simple as increasing how many points we iterate over in a serialized version of the Math Manager. The cost incurred comes from keeping more points in memory. On the graphics side, we would love to increase the size of the viewport, add translation (which would be quite simple). Finally, a goal that'd take a pretty significant amount of effort, would be improving our engine in both the Math and Graphics managers to allow us to draw the faces of triangles, requiring special math to be down to determine which faces are in front of others. This would make our project a true graphics engine as it would allow for the drawing of completely arbitrary shapes (As shapes in general are drawn as many triangles).

5 Acknowledgements

We want to acknowledge and thank our project TA, Joe who was able to provide his perspective from taking this course while debugging several sections. We also want to acknowledge Tad, who was very helpful in correcting design decisions and ensuring we did not spend too much time going down rabbit holes that lead no where with our design. We want to give special thanks to Josh who was able to debug and fix several key (and serious) issues with the Graphics Manager. We would not have been able to complete our project without the help of these individuals.

We purposely did not integrate code from ChatGPT 5 (Or any other LLM model). However ChatGPT 5 was able to identify and offer solutions that were considered and either improved upon for several key issues or ignored. Some of these issues (but not a comprehensive list of all issues) were: determining we needed to clip z when dividing, suggesting that overflow errors in the Bresenham module were causing incorrect plotting, finding variables left of sensitivity list in several FSM Controllers, and providing advice on hyperparameter values for projection math.

We took inspiration from several projects, or implementations of certain modules that we wanted to thank and give credit to. The following source material was either considered, integrated in some manner, or used to draw inspiration from:

- "Synthesizable VHDL Model of 3D Graphics Transformations", McKeon, Daniel, 2005.
  - We drew inspiration from key ideas about 3D graphics from this project, however due to some key implementation differences, we did not implement much of the VHDL from this project.
- "Drawing Lines with SystemVerilog", Edwards, Stephen, Columbia University, 2015
  - The Bresenham module was directly transcribed from SystemVerilog from these lecture notes.
- Information relating to how computer graphics works, such as rotation matrices, projection matrices, etc. was read in David J. Eck's comprehensive guide to computer graphics.

6 Conclusions

This project was a significant undertaking. With more than 5,000 lines of code written (not to mention altered, revised, reviewed, etc.) and more than a combined 180 hours of work over 2 weeks, we are both happy to be done, and relieved that it successfully operates to the expectations we set. We both thoroughly enjoyed the experience despite its difficulty. We picked something that we felt was interesting yet challenging, and set out to implement it. It was an incredibly rewarding experience both from a design and collaborative perspective, but also from an educational experience. We would certainly do this project again given the chance. We hope that learning about our design, implementation, and testing process was enjoyable. Thank you (for your patience).

Appendix A: VHDL Source Code

For each file we included, we denote the entity that is within it and the section to reference in our report. We used an addition source file, `types.vhd` that packages all user defined types that we utilized in our design.

- **vga_controller**: `vga_controller.vhd`, see section 2.1.
- **bresenham**: `bresenham.vhd`, see section 2.2.
- **bresenham_receiver**: `bresenham_receiver.vhd`, see section 2.3.
- **framebuffer**: `framebuffer.vhd`, see section 2.4.
- **graphics_manager**: `graphics_manager.vhd`, see section 2.5
- **parallel_math**: `math_manager.vhd`, see section 2.6
- **update_point**: `linalg_update.vhd`, see section 2.7
- **rotation**: `rotation.vhd`, see section 2.8
- **sine_lut**: `sine_lut.vhd`, see section 2.8
- **set_operands**: `set_operands.vhd`, see section 2.8
- **accumulate_rotation**: `accumulate_rotation.vhd`, see section 2.8
- **projection**: `projection.vhd`, see section 2.9
- **reciprocal**: `reciprocal.vhd`, see section 2.10
- **newton_lut**: `newton_lut.vhd`, see section 2.10
- **newtons_method**: `netwons_method.vhd`, see section 2.10
- **angle_dir_lut**: `angle_dir_lut.vhd`, see section 2.11
- **uart_receiver**: `uart_receiver.vhd`, see section 2.12
- **top_level_controller**: `top_level_controller.vhd`, see section 2.13

Appendix B: VHDL Testbenches

In similar fashion to Appendix A, we include the unit(s), the corresponding file, and section for simulation and hardware validation.

- **vga_controller**: `vga_testbench.vhd`, see section 3.1
- **bresenham**: `bresenham_testbench.vhd`, see section 3.2
- **bresenham_receiver**: `bresenham_receiver_tb.vhd`, see section 3.3
- **parallel_math**: `math_manager_tb.vhd`, see section 3.6
- **update_point**: `linalg_update_tb.vhd`, see section 3.7
- **rotation**: `rotation_tb.vhd`, see section 3.8
- **sine_lut**: `sine_lut_tb.vhd`, see section 3.8
- **set_operands**: `set_operands_tb.vhd`, see section 3.8
- **accumulate_rotation**: `accumulate_rotation_tb.vhd`, see section 3.8

- **projection**: `projection_tb.vhd`, see section 3.9
- **reciprocal**: `reciprocal_tb.vhd`, see section 3.10
- **newton_lut**, **newtons_method**: `newtons_method_tb.vhd`, see section 3.10
- **uart_receiver**: `uatr_receiver_tb.vhd`, see section 3.12
- **graphics_man_test_shell**: `graphics_man_test_shell_tb.vhd`, see section 3.13
- **central_controller**: `top_level_controller_tb.vhd`, see section 3.14