ECE 385

Spring 2020

Graphical Processing Unit with NIOS and SystemVerilog

Megh Shah, Ishaan Patel
Lab Section: ABC, Tuesday 11:30 AM
TA: Gene Shiue, David Zhang

1. Introduction:

At first 3D graphics may seem confusing. How can a 3D object, in this case a cube, be represented on a 2D screen? Before we go dive into the graphics pipeline, let's talk about the goal for the final project. **The goal for the final project is to be able to take a 3D blender file, and display its contents on the screen, rotating over the x-axis at a high Frames per Second (fps).**

What is a blender file?

A blender file, or more accurately an Object file, is a file that contains all the 3D information describing the object you made in Blender.

```
# Blender v2.82 (sub 7) OBJ File: ''
# www.blender.org
o Cube_Cube.002
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v 1.000000 1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 1.000000 1.000000
s off
f 2 3 1
f 4 7 3
f 8 5 7
f 6 1 5
f 7 1 3
f 4 6 8
f 2 4 3
f 4 8 7
f 8 6 5
f 6 2 1
f 7 5 1
f 4 2 6
```

A simple OBJ file describing a cube

What is Blender?

Blender is a free 3D modeling software where you can create any 3D shape/Object. It is industry standard when it comes to 3D software.

Image of software creating complex 3D scene
(Source: https://www.blender.org/)

How are 3D objects displayed on a screen?
When looking at a 3D model on a computer screen, for example a pyramid or cube, how does a GPU "render" these shapes? More importantly, how does the GPU draw a 3D depiction of the object? It doesn't make sense for the GPU to store data for each shape known to mankind, instead there must be a way to break down each shape into something common that they all have… Triangles!

Every 3D object that you see on the screen can be broken down into triangles. For example, a cube is simply composed of 12 triangles, 2 triangles per face. These triangles are called 'Primitives' and it is the input data of 3D rendering. If you looked closely, the blender file above is 3D data of 12 triangles (exactly what makes up a cube).

Graphics Pipeline (3D->2D->Screen)
Let's first go over the inputs and outputs of this whole graphics process.

Graphics Pipeline (Occurs every frame):

3D coordinates -> rotation-> projection-> 2D coordinates-> rasterization-> pixel color-> frame buffer -> output screen

3D coordinates:
The first step is to have a 3D representation of your object. In this case, our blender file of a cube will do.

Rotation:
Next is to "rotate" our coordinates. For our renderer we limited rotation to only the x-axis. The way to "rotate" is to multiply every coordinate matrix by a rotation matrix.

Rotation Matrix over x-axis:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

Notice the trigonometric functions

How can we represent decimal values given 32-bit coordinates(32-bit for each x,y,z)?
How can we calculate trigonometric functions once we figure out how to do decimals?

The answer is: Fixed-Point Notation.

An easy way to represent a decimal value is by using fixed-point notation.
Fixed-point notation is where a specific number of bits will be “fractional” and “Integer” bits. In our case, we chose to do 32-bit, signed fp notation.

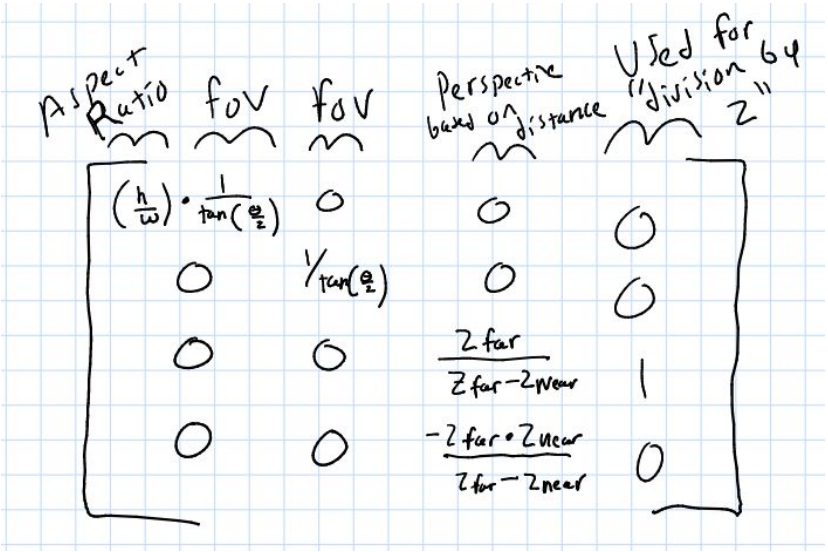
32-Bit Signed Fixed-Point Notation

Sign Bit [31]	Integer Bits [30:15]	Fractional Bits [14:0]

Trigonometric Functions
Once the representation of decimals was determined, trig functions aren’t calculated by the GPU, instead a RAM containing each function’s values is used. The GPU can simply “look-up” the desired angle value to a certain level of precision. These values are in Fixed Point notation and are generated by a custom c-program we created.

Projection
The core of a 3D engine is projection. Projection is when the GPU converts the rotated 3D points into a 2D plane(what the computer screen is). Each 3D point is multiplied by the projection matrix which will produce the desired 2D values.

Projection Matrix:



Rasterization

This step involves us computing the triangle images on the screen. More specifically, what are the pixel coordinates in between the 2D vertices that we got from projection? The way we found the coordinate values is by using Bresenham's line algorithm. However, Bresenham's line algorithm only works for slopes less than 1 and that are positive. This wouldn't work for the generality of our GPU, thus, we heavily modified it so that we can generate coordinates between any two points.
(Link for more info: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm)

Pixel Color

Because our focus was on the Frames per Second as well as blender file integration, we decided to do a wire frame design. Thus, instead of incorporating a shading buffer, all we do is produce a pixel color to all the coordinates generated in the rasterization process.

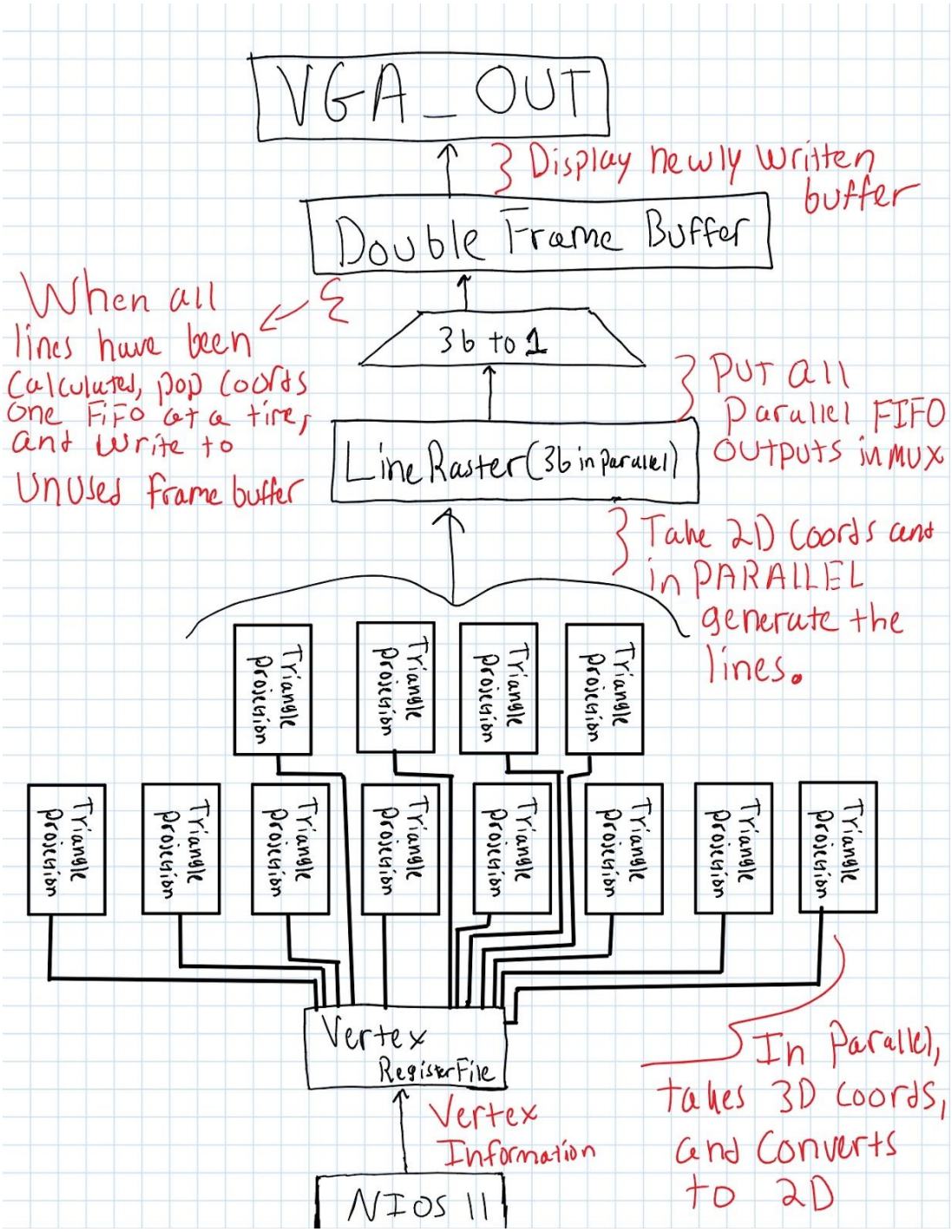
Frame Buffer

Modern GPUs use more than one frame buffer. The reason for this is because while the display is reading from one buffer, the GPU can write to another without any vertical or horizontal screen tearing. We embedded this concept into our design and incorporated a double buffer. One buffer is being read by the VGA controller while the other one is being written to by the GPU.

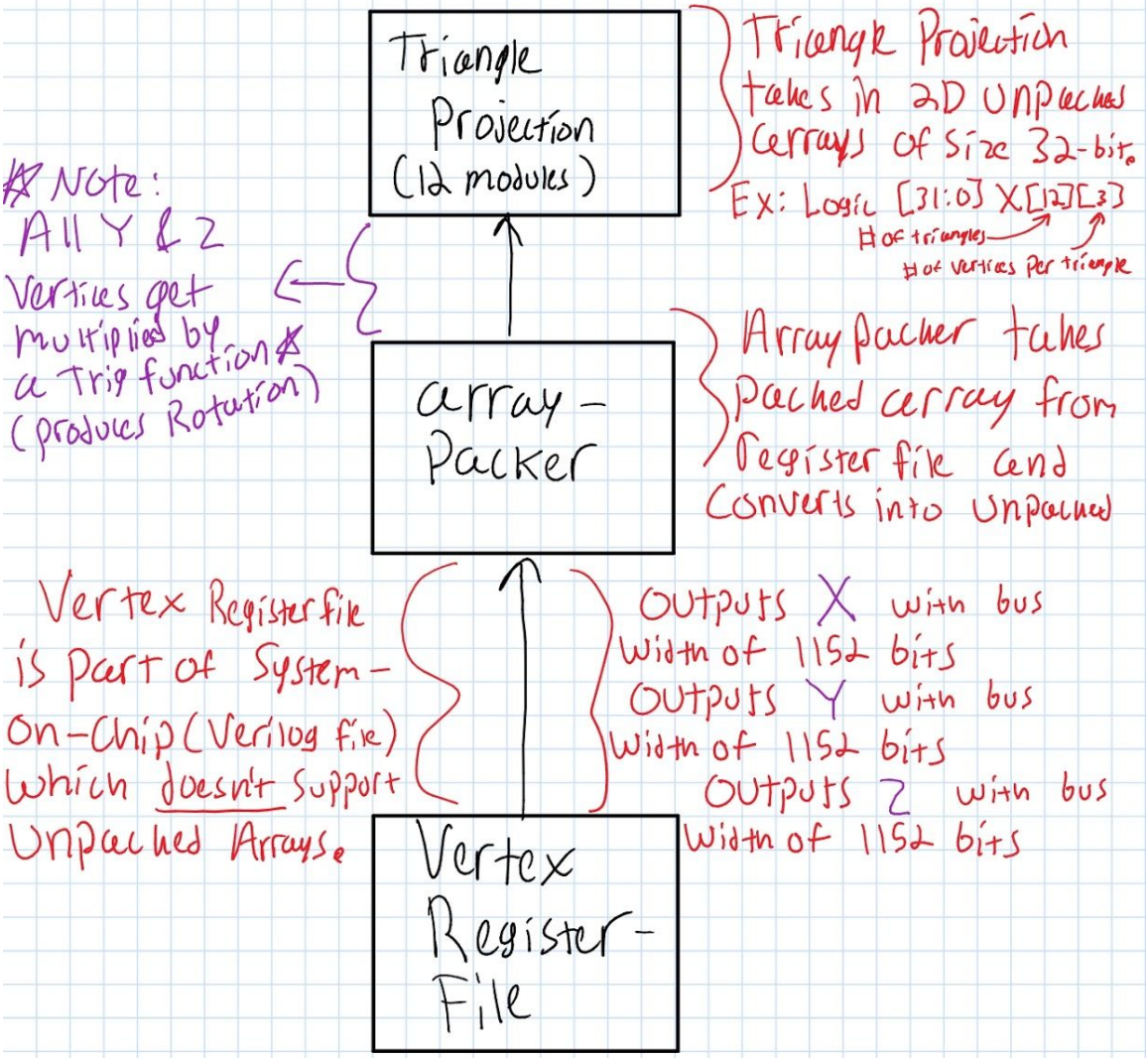
Output Screen

The screen will show the contents of the frame buffer not being used by the GPU and a rotated blender file object should appear on the screen.

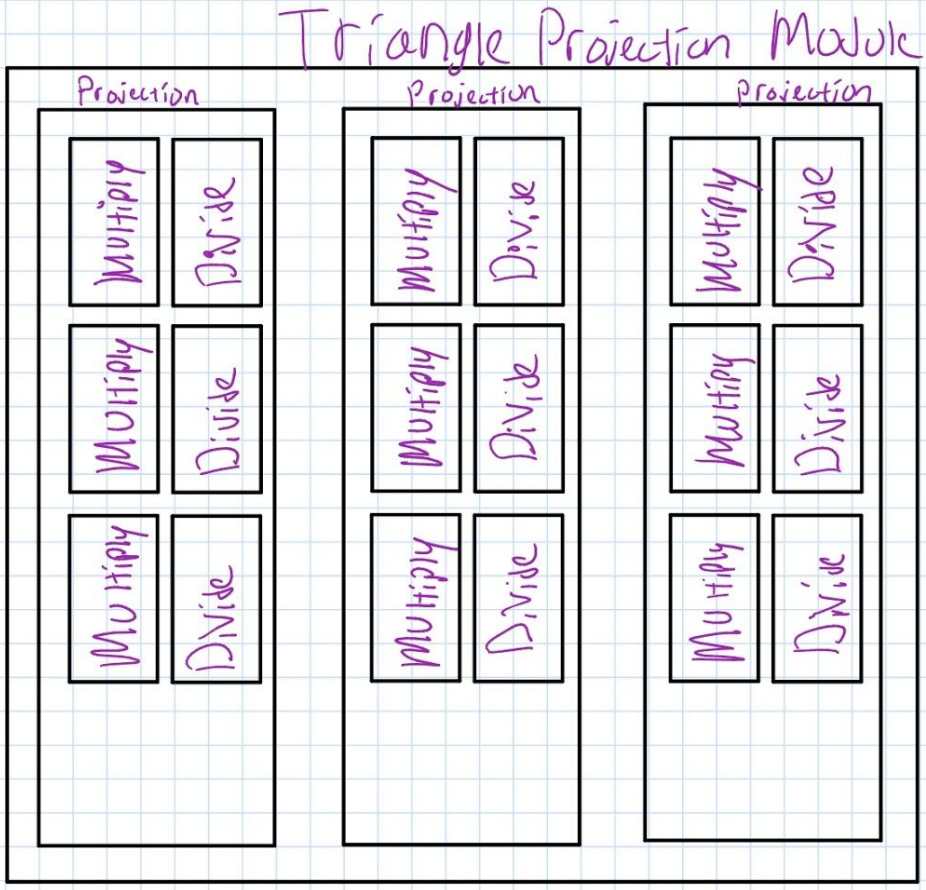
BLOCK-DIAGRAMS:



In between Vertex Register File and Triangle Projection.

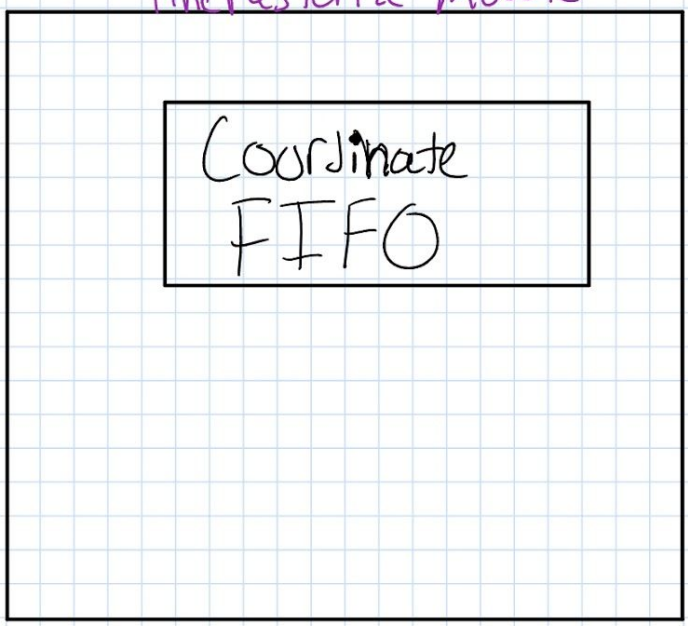


Inside Triangle Projection



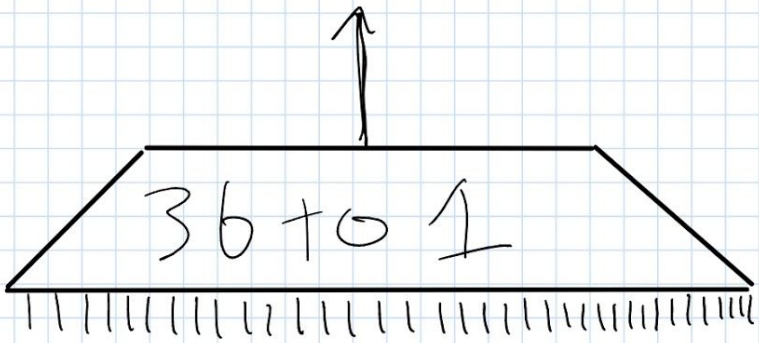
A triangle projection module (we have 12) converts 3D vertices of a single triangle into 2D space. We then take the 2D vertices for Rasterization.

Rasterizing Triangles/Lines



Each 2D vertices produced from the projection(s) module(s) are inputes to their respective lineRasterizer module. The line rasterizer module generates all the coordinates between two vertices and stores them in a fifo.

36 TO 1 MUX



Each Line rasterize module is an Input to the mux. Once all modules are finished (can vary) a FSM will pop Coordinate Data One Fifo at a time into the Unused Frame buffer.

How is Rotation Done?

-Sin(θ) Memory	Sin(θ) Memory	Cos(θ) Memory

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To achieve rotation, the 3D coordinate must be multiplied by a Sin, -Sin or Cos value before projected into 2D Space. To get the Trig Value, we store the values (to a certain precision) in BRAM as Memory.

2. Summary of Operations:

The graphical processing unit begins its process on the software side. The C++ program reads a .obj file which is a text file that contains the number of vertices, faces, and triangle coordinates. The triangle coordinate values, which are three-dimensional, are read and stored into a register file which consists of 128 registers and resides within the NIOS II processor. Once the start register is set to "1" by the C++ program, which it does after it finishes writing the values into the register, the graphical processing unit begins the rendering of the object. First, the three-dimensional coordinates are converted and projected onto a two-dimensional screen. Then, a line is drawn between these coordinates to complete the triangle. This process is repeated until every triangle is drawn and in the correct position. This will create an object that is made up of many different triangles, a method used by many high-end GPUs. The object is then rotated around a selected axis to show that it is fully rendered and indeed three-dimensional.

3. Detailed Description and Diagrams of Operation:

3.1 NIOS II Processor Configuration:

The NIOS II processor contains many different cores. Aside from the basic cores, such as SysID, JTAG, and SDRAM, we included a new IP core. This IP core, called registers_0, was designed to function as a register file, similar to how to avalon_aes_interface.sv of lab 9. The register file contains 128 32-bit registers. The first 108 registers are reserved to hold the values of the three-dimensional coordinates. Each coordinate, such as X, Y, or Z requires 32-bits as it is represented using fixed-point notation. Fixed-point notation is similar to floating-point where there are integer and fractional bits. However, the main difference with fixed-point notation is that it is easier and more convenient for humans to use and read. In 32-bit signed fixed-point representation, the most significant bit is the sign bit. The 16-bits that follow after are used to represent the integer value and the remaining 15-bits are used to represent the fractional part.

32-Bit Signed Fixed-Point Notation

Sign Bit [31]	Integer Bits [30:15]	Fractional Bits [14:0]

Registers zero through thirty-five hold up to 36 x-coordinates whose values which are all represented using 32-bit signed fixed-point notation with one x-coordinate in one register. Registers thirty-six to seventy-one hold up to 36 y-coordinates in the same format, and registers seventy-two to one-hundred-seven hold up to 36 z-coordinates in the same fashion. Register one-hundred-twenty-six holds the bit that signals the hardware to begin the render process, and is respectively denoted as the start register, and register one-hundred-twenty-seven holds the number of triangles that the coordinates in the registers above make up. Alongside the registers, the IP core contains several I/O signals to control the access of these registers. The IP core contains:

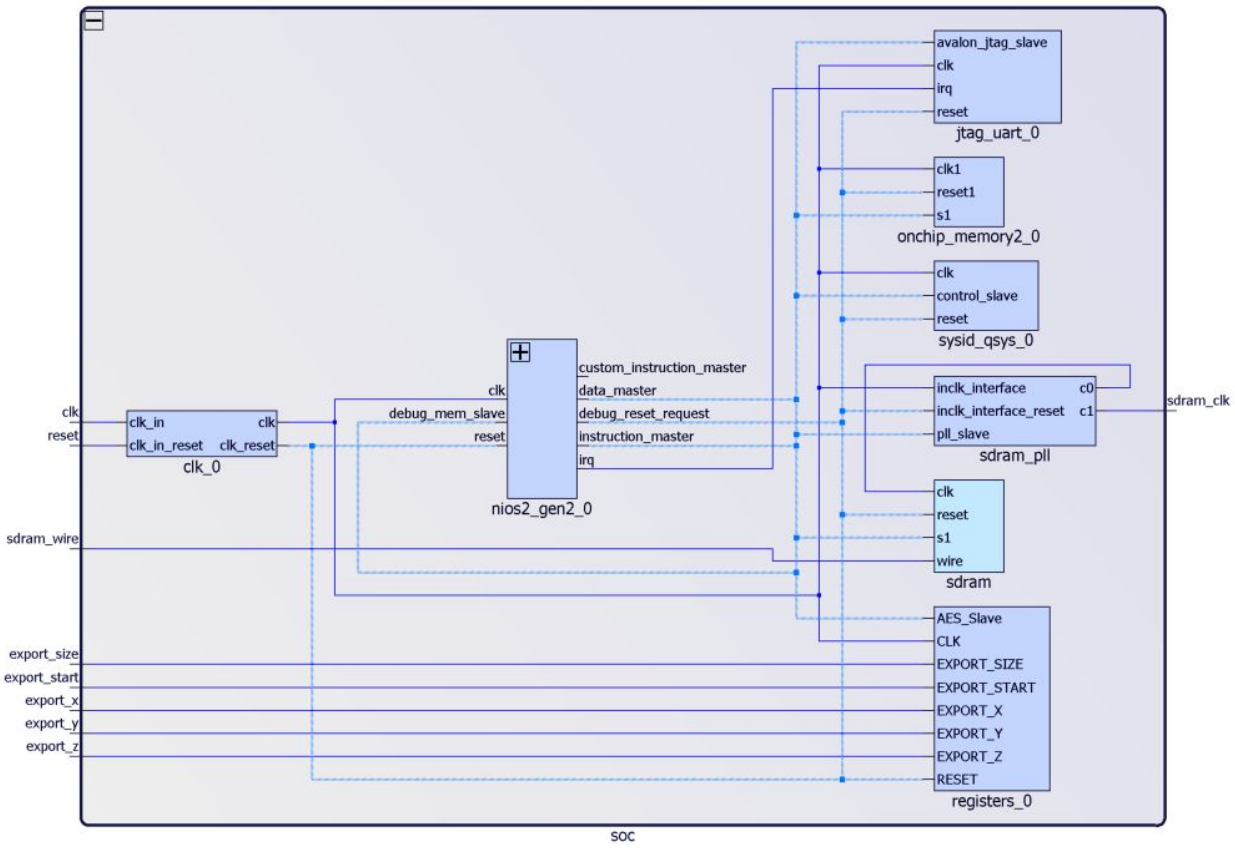
INPUT:

- **CLK** - A basic clock from the NIOS II processor
- **RESET** - A reset signal controlled by the NIOS II processor
- **AVL_READ** - A read signal that allows registers to be read when high
- **AVL_WRITE** - A write signal that allows registers to be written when high
- **AVL_CS** - Chipselect signal that allows registers to be written or read when high
- **[6:0] AVL_ADDR** - Seven-bit address signal that is used to select which register we are reading or writing into
- **[31:0] AVL_WRITEDATA** - The 32-bit data we want to store in the register

OUTPUT:

- **[1151:0] EXPORT_X** - 1152-bits which consist of the 32-bit datas in registers 0 through registers 35 which hold x-values. All 32-bit are combined back-to-back
- **[1151:0] EXPORT_Y** - 1152-bits which consist of the 32-bit datas in registers 36 through registers 71 which hold y-values. All 32-bit are combined back-to-back
- **[1151:0] EXPORT_Z** - 1152-bits which consist of the 32-bit datas in registers 72 through registers 107 which hold z-values. All 32-bit are combined back-to-back
- **[31:0] EXPORT_START** - Holds a “1” when registers are done being written into by the C++ program
- **[31:0] EXPORT_SIZE** - Holds the number of triangles that the coordinates make

Many of the I/O signals are used and controlled by the NIOS II processor. However, there are certain signals that are exported, such as EXPORT_X, EXPORT_Y, and EXPORT_Z, from the IP Core to be used with other files. The EXPORT X,Y, Z signals are exported conduits as they contain the values of the x, y, and z coordinates. These signals go into other modules, such as the projection module, so that arithmetic can be applied on these values. Below, the schematic for the NIOS II SOC is provided.


avalon_jtag_slave
clk
irq
reset
jtag_uart_0
clk1
reset1
s1
onchip_memory2_0
clk
control_slave
reset
sysid_qsys_0
custom_instruction_master
clk
data_master
debug_mem_slave
debug_reset_request
reset
instruction_master
irq
nios2_gen2_0
inclk_interface
c0
inclk_interface_reset
c1
pll_slave
sdram_pll
sdram_clk
clk
reset
clk_in
clk
clk_in_reset
clk_reset
clk_0
clk
reset
s1
wire
sdram
sdram_wire
AES_Slave
CLK
EXPORT_SIZE
EXPORT_START
EXPORT_X
EXPORT_Y
EXPORT_Z
RESET
registers_0
export_size
export_start
export_x
export_y
export_z
soc

3.2 C++ Program:

The main purpose of the C++ program is to take in coordinates and store them into the registers in the register file in the NIOS II processor, so that other modules can access these coordinate values. The C++ program works by reading a blender file. A blender file is also known as an object file and has the extension .obj. These files are essentially test files that contain information about coordinates, such as the x, y, and z values and how many coordinates there are. Below is an object file for a unit cube.

```
# Blender v2.82 (sub 7) OBJ File:
# www.blender.org
o Cube_Cube.002
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v 1.000000 1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 1.000000 1.000000
s off
f 2 3 1
f 4 7 3
f 8 5 7
f 6 1 5
f 7 1 3
f 4 6 8
f 2 4 3
f 4 8 7
f 8 6 5
f 6 2 1
f 7 5 1
f 4 2 6
```

The lines that begin with a “v” or a “f” indicate either a vertex or a face. The C++ program reads the lines that begin with a “v” and stores the x, y, and z coordinates of that vertex into vectors. When it reads a line that begins with a “f”, it refers to the vertices with the numbers following the “f”, and finds those vertices x, y, and z values. With these values, the C++ program then converts the decimal value into fixed-point notation. To do this, we simply multiply the value by 2^15, since we have 15 fractional bits, and then round the value. We then sign extend it to 32-bits. After this, the program stores the fixed-point value into the corresponding registers in the register file. The C++ program is able to access these registers as it contains a pointer to the base address of the IP core. From there, all the C++ program has to do is index the pointer to access a specific register as if it were indexing an array or vector.

```
volatile unsigned int * AES_PTR = (unsigned int *) 0x10001200;
```

Once the entire file is read, the program will set the size register to represent the number of triangles and the start register to a “1”. The start register being a “1” signals the end of the software portion of the GPU and begins the hardware part.

3.3 Projection:

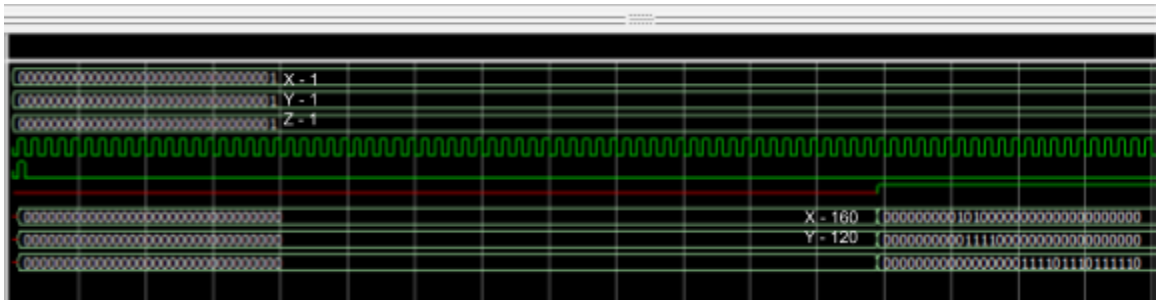
Projection is the first step in the hardware process of the GPU. The sole purpose of the projection module is to take points from three-dimensional space, and essentially “project” them, in other words convert them into two-dimensional space, such as a monitor. After the C++ program reads the coordinates and stores them into their respective registers, the 1152-bit datas holding the x, y, and z values are repacked into unpacked double arrays. The first index of these arrays represent the triangle number while the second index represents which coordinate, such as x, y, or z, of the triangle we are referring to. The projection module takes in an entire triangle which means it takes in 3 vertices or 9 coordinates. The module then applies the steps of the Perspective Projection Process to convert the coordinates into two-dimensions. Listed below are the steps.

Perspective Projection Process:

1) **Multiply the x-coordinate by the aspect ratio (height / weight)**
2) **Multiply both the x-coordinate and y-coordinate by 1/tan(Θ) where Θ represents the angle at which we are viewing the screen**
3) **Multiply the z-coordinate by Zfar / Znear, which is the furthest distance and nearest distance the user wants the object to be in the screen**
4) **Subtract the z-coordinate by Zfar / Znear**
5) **Divide all three coordinates by the original z-coordinate**
6) **Add one to the x-coordinate and y-coordinate and then divide by two**
7) **Multiply the x-coordinate by the width and the y-coordinate by the height**

As detailed by the steps above, the three-dimensional coordinates from the registers undergo the process above. The perspective projection process is a form of matrix multiplication as demonstrated at the top in the introduction section. The resulting x-coordinate and y-coordinate can be used to plot the three-dimensional point on two-dimensional space where the z-coordinate will make the points appear bigger or smaller to imitate the point being closer or further from the screen. These two-dimensional coordinates are then passed onto the line rasterizer module. Below is

attached a simulation of the projection module. It showcases transforming the point (1, 1, 1) into (160, 120) to represent a three-dimensional coordinate into two-dimensional space. These given inputs and outputs are represented in 32-bit signed fixed point representation.



3.4 Line Rasterization:

After projection transforms the three-dimensional coordinates into two-dimensions, the GPU has a bunch of vertices and points. With these points, the GPU can utilize a modified version of Bresenhem's Line Algorithm to determine the points between two vertices and connect them together to create a line. This is the purpose of the line rasterizer. This module takes in two different points. With these points, it applies the following algorithm to produce several points that fall between these two points:

Brensenham's Line Algorithm:

1) **Take in two inputs: (x1, y1) and (x2, y2)**
2) **Calculate the difference between the x-coordinates and the difference between the y-coordinates**
3) **Using the differences, step to the next point by multiplying the difference with the original start coordinate**
4) **If the next pixel falls within >= 0.5 tolerance of the algorithm's expected pixel, that pixel will be part of the line. Otherwise, continue**

Using this algorithm between two points, the line rasterizer module is able to compute many points between the two inputted points, which when all connected together will produce a line. These points are stored in a FIFO structure. FIFO stands for first-in-last-out meaning whatever coordinates are stored in the structure last, will be the first to be read and accessed when accessing the structure. Since the GPU is capable of taking in 36 triangles, the GPU is also then capable of creating 36 lines all of which have

their own FIFO structure in which they hold their points. After each line is done being computed and its FIFOs are filled, the module signals to the output to begin loading in the frame buffers.

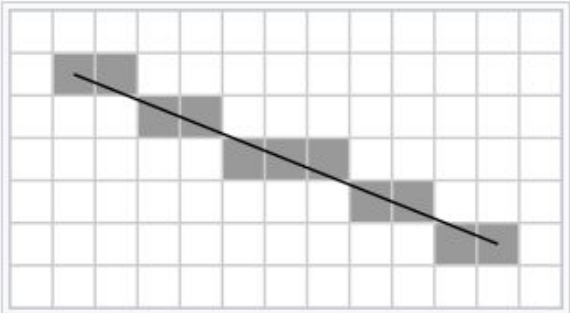


Illustration of the result of Bresenham's line algorithm. (0,0) is at the top left corner of the grid, (1,1) is at the top left end of the line and (11, 5) is at the bottom right end of the line.

3.5 Frame Buffers and VGA output:

Once the FIFO structures are filled and contain all the points needed to compute a line, the frame buffers are ready to be loaded. The purpose of a frame buffer is to take in the points or coordinates that need to be colored white for that specific frame. The GPU contains two frame buffers. While the frame buffer for one frame is being used by the GPU to color the coordinates on the screen, the other frame buffer is being written into by the other modules. This is to increase the speed and frame rate of the GPU as when one frame is done being loaded and is displayed on the screen, the other frame will already have the next coordinates to be colored ready to go. A 2-to-1 MUX with the inputs being the two frame buffers and the output being the VGA controller is utilized to switch between the buffers as its select line is constantly flipped to alternate between the two buffers. A 36-to-1 MUX connects the 36 FIFO structures which contain all the points that need to be colored and connects it to the frame buffer that is not displaying its points on the screen. Once the frame buffer is filled with the coordinates that need to be colored in to produce a line, and ultimately triangles that represent an object, the VGA controller takes each coordinate value as 18-bit datas. The first 9-bits represent that value of the x-coordinate while the remaining 9-bits represent the value of the y-coordinate. When the VGA controller advances the electron beam and reaches the pixel whose x-coordinate and y-coordinate are the same as the values of the coordinate in the frame buffer, that pixel is colored white. This process repeats until all coordinates are colored in.

Above is a picture of a cube drawn by the GPU after all the above steps are completed.

3.6 Rotation:

At this point the GPU is able to render frames of an object. This means that it is able to take in points and put them together to create an object on the screen, and because of the frame buffers, it is able to update the image frequently. However, In order for the object to rotate and move, the GPU must be able to quickly change the coordinates and constantly change them so that the lines and triangles produced are in different positions and different angles. This is the purpose of rotation. Rotation takes in the three-dimensional points from the register module, and multiplies those points with values of sine and cosine for every new frame. The sine and cosine values are precomputed and are stored in memory. This is to increase the speed and efficiency of the GPU since these values are constantly used. Whenever a sine or cosine result is needed, the GPU simply uses a lookup table and finds the precomputed value to multiply the coordinate by. When multiplying a coordinate value by sine or cosine, it will change the coordinate position and will look like the coordinate is being viewed from another angle. The GPU repeats this process with all coordinates, so in the end, the entire object will be rendered from a different angle. Each frame is calculated with different values of sine and cosine meaning that everytime the frame buffers are read and color in pixels, the resulting image will be slightly shifted and angled differently. When all this is performed at fast speeds, we see a smooth animation of a cube, or the object that is being rendered, rotating about a certain axis.

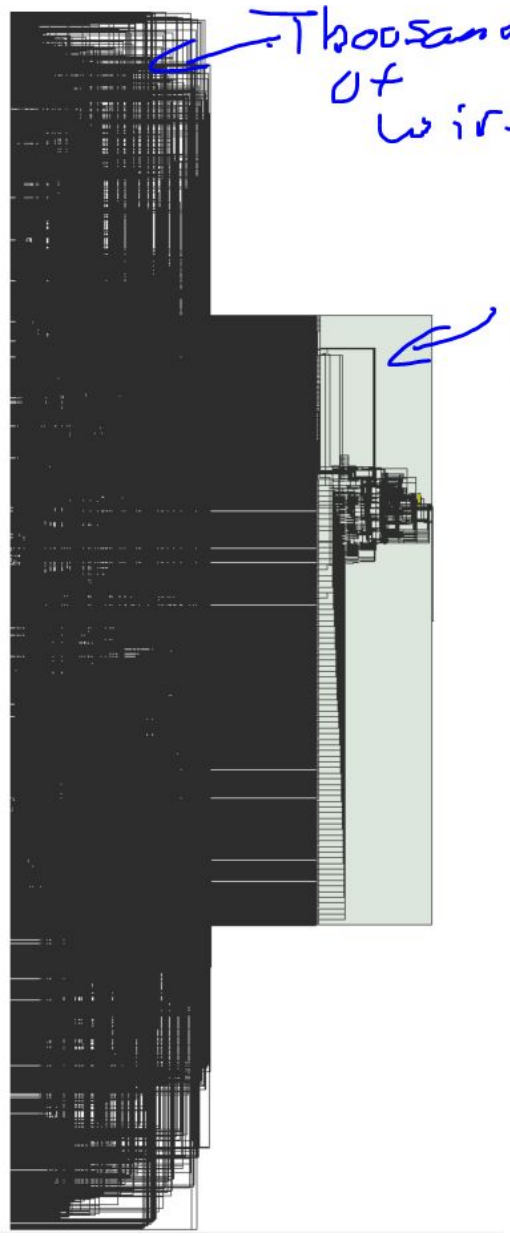
```
-- Quartus Prime generated Memory Initialization File (.mif)
WIDTH = 32;
DEPTH = 315;
ADDRESS_RADIX = UNS;
DATA_RADIX = BIN;
CONTENT
BEGIN
0 : 00000000000000000000000000000000;
1 : 00000000000000000000000001010001111;
2 : 00000000000000000000000010100011110;
3 : 00000000000000000000000011110101101;
4 : 00000000000000000000001010000111011;
5 : 00000000000000000000001100011000111;
6 : 00000000000000000000001111010010011;
7 : 00000000000000000000010001111011101;
8 : 00000000000000000000010100011100101;
9 : 00000000000000000000010111011101010;
10 : 00000000000000000000011001011101110;
11 : 00000000000000000000011011111101111;
12 : 00000000000000000000011111001101101;
13 : 00000000000000000000100000011101000;
14 : 00000000000000000000100011011100000;
15 : 00000000000000000000100101110101000;
16 : 00000000000000000000101000010000100;
17 : 00000000000000000000101010101100000;
18 : 00000000000000000000101101000010111;
19 : 00000000000000000000101111011111010;
20 : 00000000000000000000110001110110000;
21 : 00000000000000000000110100001100001;
22 : 00000000000000000000110110100000101;
23 : 00000000000000000000111000110100011;
24 : 00000000000000000000111011000111000;
25 : 00000000000000000000111101010111110;
```

The image to the left showcases how the lookup tables are used. The image represents the sine lookup table. Whenever a sine operation is required, the value is used to index the lookup table, and the corresponding result, which is represented in 32-bit signed fixed-point notation, is returned.

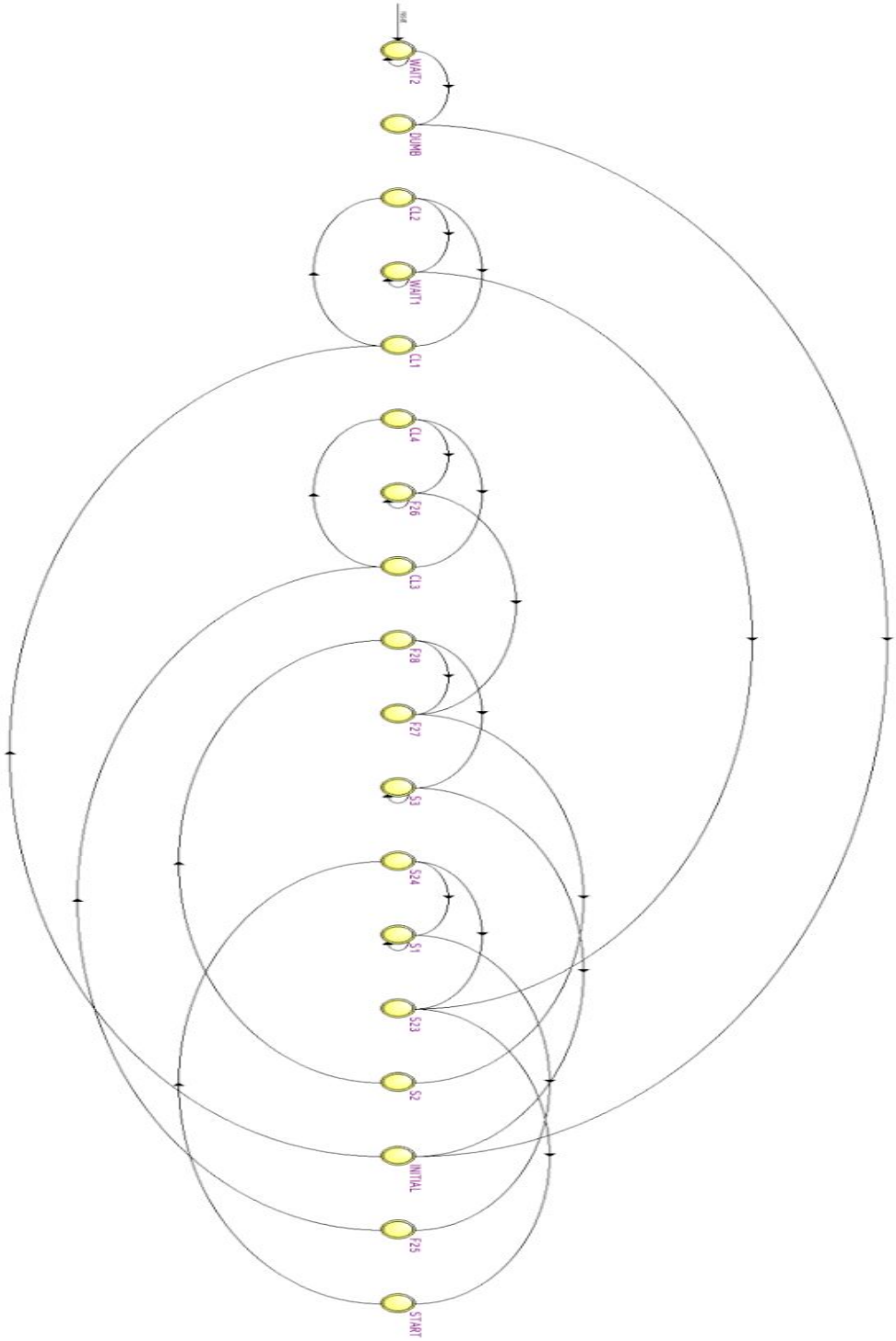
4. State Machines & Generated Block Diagrams:

Due to high throughput of GPU, we can't show the RTL view of any design since they are simply too big for the viewer to show. Please refer to the [hand-drawn block diagrams.]

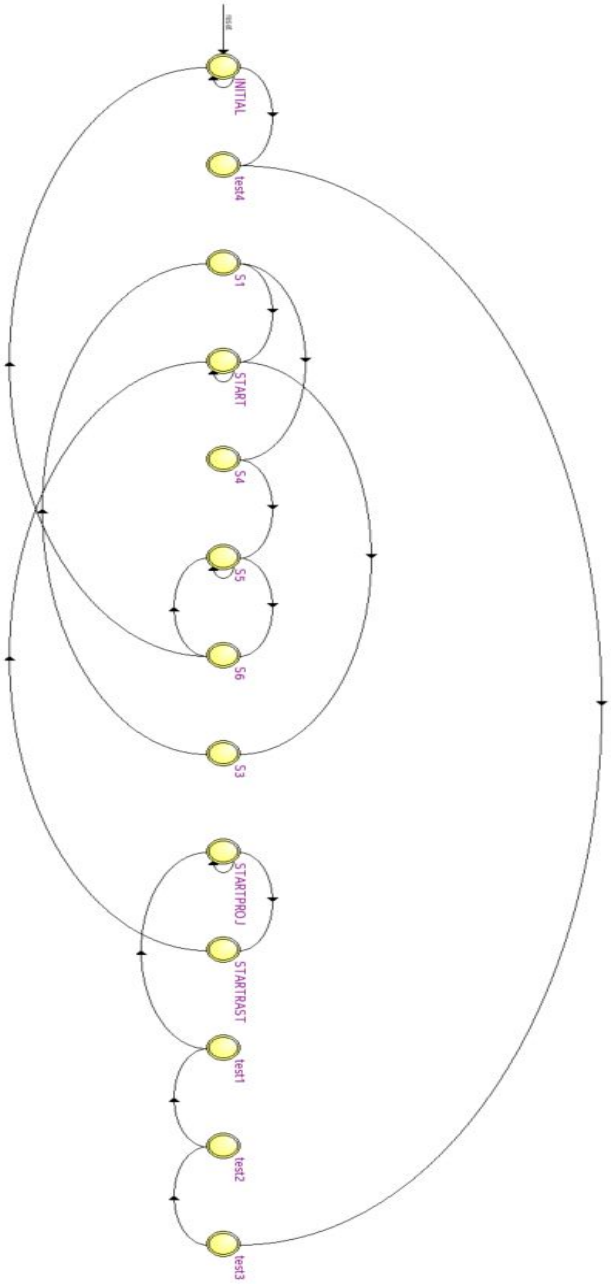
Example:



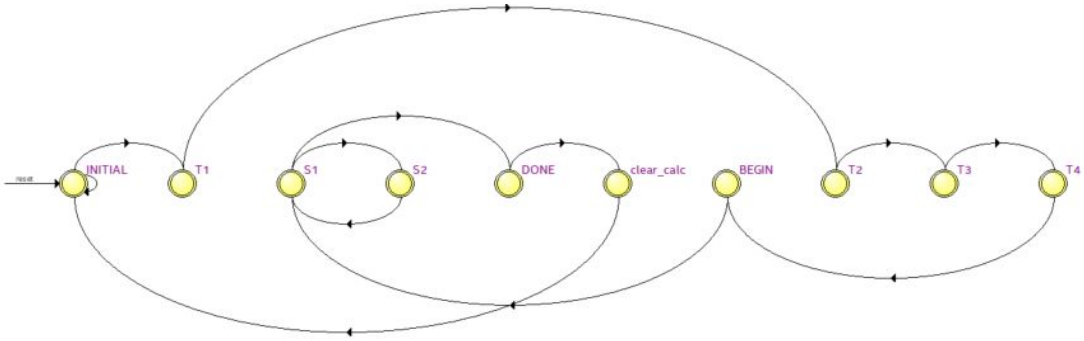
4.1 State Diagram of top.sv (Reads and Writes to Frame Buffer)



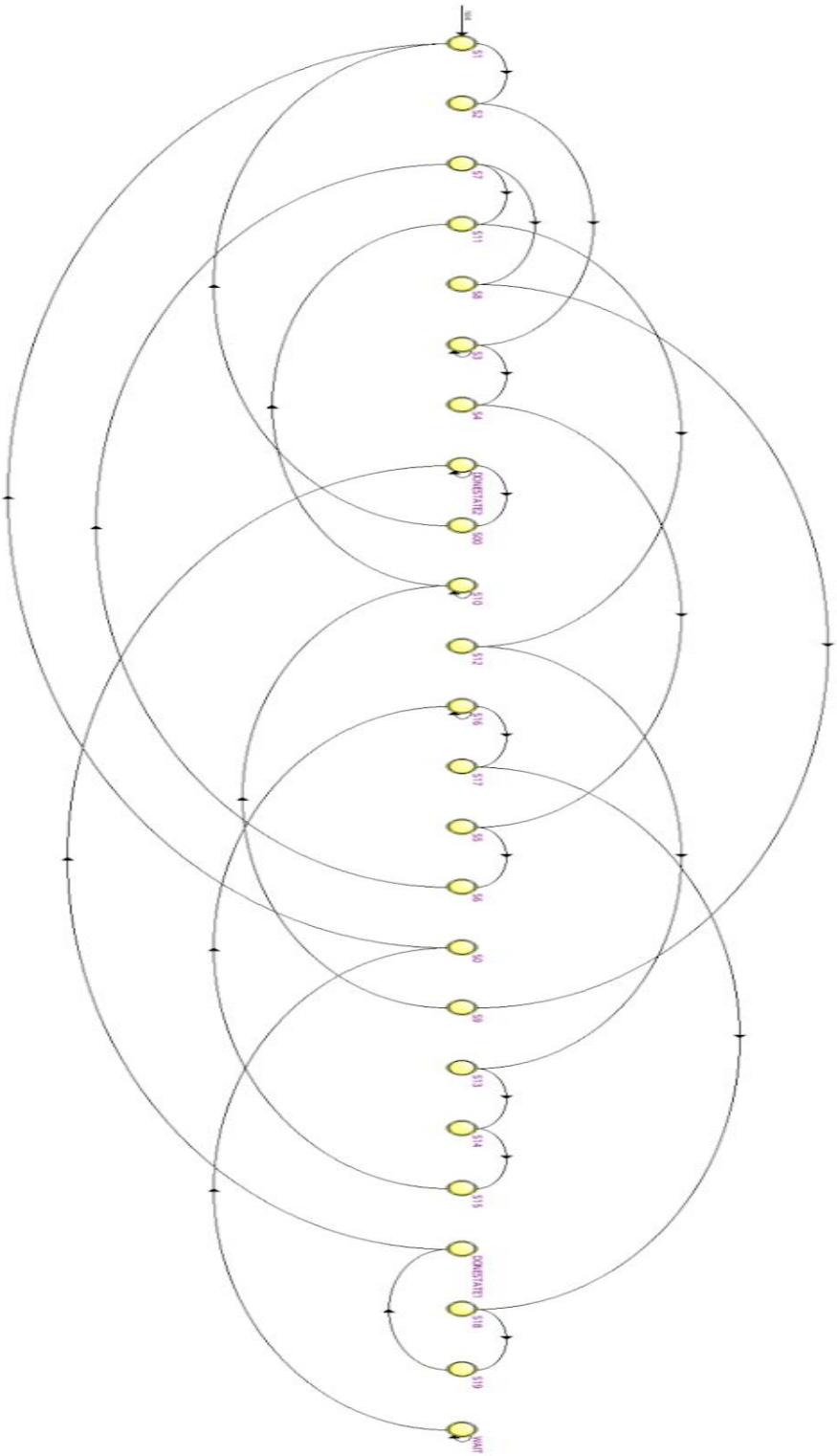
4.2 State Diagram of polygonRaster.sv (Projection & Rasterization)



4.3 State Diagram of linerasterize.sv (Rasterizes Lines)



4.4 State Diagram of projection.sv (Converts 3D Pixels to 2D Pixels)



5. Modules & Usage Information:

5.1 Modules & .sv File Descriptions

HexDrivers.sv - Used to display debug data on the HEX displays on the FPGA board

VGA_controller - contains timing information on when HS or VS are active and DrawX and DrawY. We use DrawX and DrawY to access memory buffers.

vga_clk.v - produces 25 Mhz clock for VGA signal using PLL

SDRAM Controller – Contains instructions which the NIOS II processor executes from

SDRAM Clock – Special clock used for the SDRAM to avoid metastability and propagation delays

On-Chip Memory – Used to store data and values. Needs to be constantly refreshed or else will lose data

System ID Checker – Checks whether the hardware and software are compatible

jtag_uart.sv - Used as a communication link between NIOS and Computer.

NIOS_2.sv - NIOS CPU, can use C/C++ code

registerFile.sv - Stores all the 3D vertices that NIOS puts

top.sv - Top Level, interfaces with VGA and SDRAM and contains FSM for Frame management.

arrayPacker.sv - Takes 1152-bit wide bus for X,Y,Z and organizes as a 2D unpacked array.

output_screen.sv - contains VGA_Controller and CLK and produces VGA signals. Also contains the double buffer and output controls to it where top.sv accesses it.

GBUFFER1.qip - Intel IP on chip memory. Contains 3-bit value for each RGB pixel.

GBUFFER2.qip - Intel IP on chip memory. Contains 3-bit value for each RGB pixel.

linerasterize.sv - Given 2 vertices in any position, produces (x,y) coordinates that form a line between said vertices. The coordinates are stored in a FIFO that is accessible to a top level.

lineoutput.qip - Intel IP on chip FIFO. Contains xy coordinates that need to be displayed on screen.

polygonRaster.sv - Generates 12 render cores (projection,rasterization), each running in parallel. The output are the xy coordinates that top.sv will read and fill in with color.

var_mux - Custom variable mux where you can specify inputs,width and size of select line. This is variable because the number of render cores can easily be decreased/increased. The input of this mux is the output of linerasterize.sv fifo's and the output of the mux is what top.sv uses to fill in the frame buffer.

triangleProjection - takes in 3 3d vertices and produces 3 2d vertices. This is simply a wrapper for 3projection modules. The output of this module goes to the linerasterizer.

projection - Takes in a 3d vertex and produces a 2d vertex. Matrix math is involved and 3 divider modules and 3 multiplication modules are used since 32-bit numbers are in fixed point notation.

divider- This module is used for fixed-point division and is by Freecores.
CREDIT: https://github.com/freecores/verilog_fixed_point_math_library

multiplier - This module is used for fixed-point multiplication and is by Freecores.
CREDIT: https://github.com/freecores/verilog_fixed_point_math_library

sin.qip - Intel IP on chip memory that stores 314 values of sin from 0 to 2pi. The values are stored in fixed-point notation and were created by a custom c-program that we made which converts float to fixed.

cos.qip - Intel IP on chip memory that stores 314 values of cos from 0 to 2pi. The values are stored in fixed-point notation and were created by a custom c-program that we made which converts float to fixed.

neg_sin.qip - Intel IP on chip memory that stores 314 values of neg_sin from 0 to 2pi. The values are stored in fixed-point notation and were created by a custom c-program that we made which converts float to fixed.

5.2 Resource Usage Block:

LUT	81,187
DSP	532 Multipliers
Memory (BRAM)	976,896
Flip-Flop	29,802
Frequency	58.63 Mhz
Static Power	114.44 mW
Dynamic Power	571.71 mW
Total Power	760.97 mW

6. Conclusion:

6.1 Errors, Bugs, and Issues:

When designing the GPU, we knew that we were going to run into many different errors and bugs due to GPUs being complicated. One error we ran into was in the early parts of our code and it was regarding projection. For projection, when applying the algorithm we were required to use many division and multiplication operators. However, the usage of the division operator significantly slowed down our compilation time as the division chip on the FPGA board requires many clock cycles to complete. Due to this, we suffered metastability issues as our state machine would move on without the division operation completing. Furthermore, we also had issues with the multiplication operator. Since we were computing a lot of multiplication operations, the FPGA board utilized all 532 on-board multipliers. This led to our code further being slowed down due to it having to wait until the previous multiplication operations function for the next ones to start. We resolved this issue by using a fixed-point division and multiplication modules which were provided online by Freecores. Using these modules, we were able to significantly cut down the compilation speed and increase the frames per second. Another issue we ran into was the usage of fixed-point notation. In many parts of our code, we faced issues regarding bits such as overflow, extending registers with the wrong sign, or simply not sign extending at all. Issues such as these caused our object that was rendering to have many bugs such as lines or points being off screen or too big, because it was reading the negative value as a really big positive number, or points simply not being rendered due to the registers not being sign extended and were less than 32-bits. However, with careful testing and debugging we were able to locate these issues and put precautions, such as manually sign extended when SystemVerilog or C++ would not. Overall, other than various other small issues, we were able to complete this project as we intended due to careful preparation and research.

6.2 Conclusion:

When we began this project, our goal was to create a GPU that is able to read an object file and produce coordinates and lines to reproduce that object and rotate it. In the end, we believe we were able to complete this goal regardless of the many issues we ran into. We were able to complete many different modules and algorithms, a C++ program, NIOS implementation, and object files and were able to coherently utilize all of these together to create a functioning GPU. In this project, we believe we demonstrated all the skills that we learnt in this class and previous labs. From creating IP Cores in NIOS and modules, to implementing a bridge between software and hardware, and utilizing external peripherals and I/O devices, this lab represents our work and knowledge of ECE385.