EE 705 (Spring'22)

VLSI Design Lab
Project Report

Indian Institute of Technology, Bombay

Project Topic:

3D Object Rendering using FPGA

Team Members:

1. Surbhika Rastogi (213070056)
2. Hitesh Kumar Sahu (213079013)
3. Rajat Kumar Panigahi (213070057)

Table of Contents

1. Introduction

Our project deals with hardware implementation of 3D rendering pipeline which is extensively required in gaming and 3D simulations. 3D rendering is the process in which a 3D model is converted to 2D image and as and when we rotate or move the 3D model we will get a 2D image such that it will appear as 3D and move in our desired direction. This concept is very essential in Gaming in which the environment changes according to the point of reference (you).

In this project we implement a 2D image from the 3D model. This will be done using hardware. Direct hardware implementation of the renderer is faster than software implementation. Rendering is a very time-consuming process so, using hardware implementation would work faster.

The input to this pipeline is a commonly used .obj format 3D object file. An OBJ file contain vertex data, free-form curve/surface attributes, elements, free-form curve/surface body statements, connectivity between free-form surfaces, grouping and display/render attribute information. The most common elements are geometric vertices, texture coordinates, vertex normals and polygonal faces. Since this project demonstrates proof of concept as part of course project only and not a practically deployable implementation, so we have consumed only the vertices and faces information from the file which is stored in following format in the file.

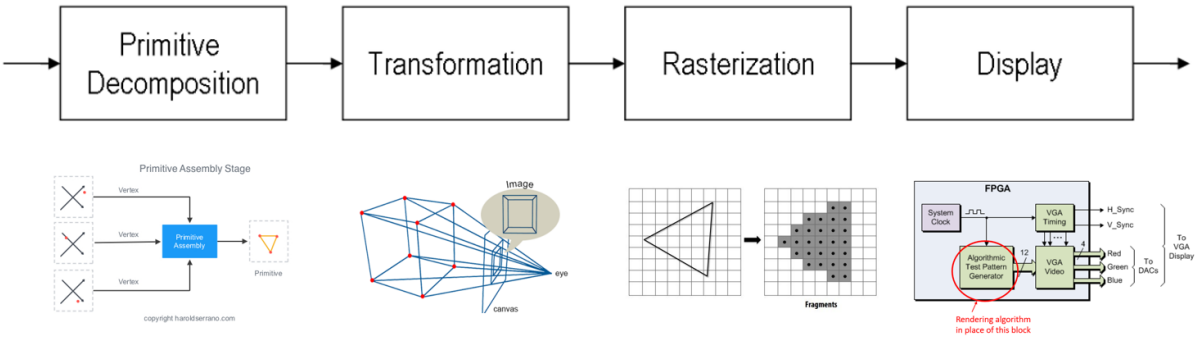
List of geometric vertices, with (x, y, z [,w]) coordinates, w is optional and defaults to 1.0.

```
v 0.123 0.234 0.345 1.0
v ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
```

Section 2 contains Block-Level Design. Section 3 contains the software Implementation. Section 4 contains Hardware Implementation. Section 5 contains challenges and conclusions.

2. Block Level Design

The block level design is shown in the figure below:



2.1. Modules and their associated theory

2.1.1. Primitive Decomposition

A 3D object is defined using its vertices in a 3D coordinate system. These vertices define the boundary of the 3D object, whether it's a simplest polygon or a complex machine gun model. The complex models are decomposed into basic shapes such as triangle or quadrilateral commonly for further processing. This is called primitive decomposition.

In our project implementation, we have developed this part in Python. We pre-process the 3D model in Python to decompose each face of the model into one or more triangles which are further fed to the transformation block.

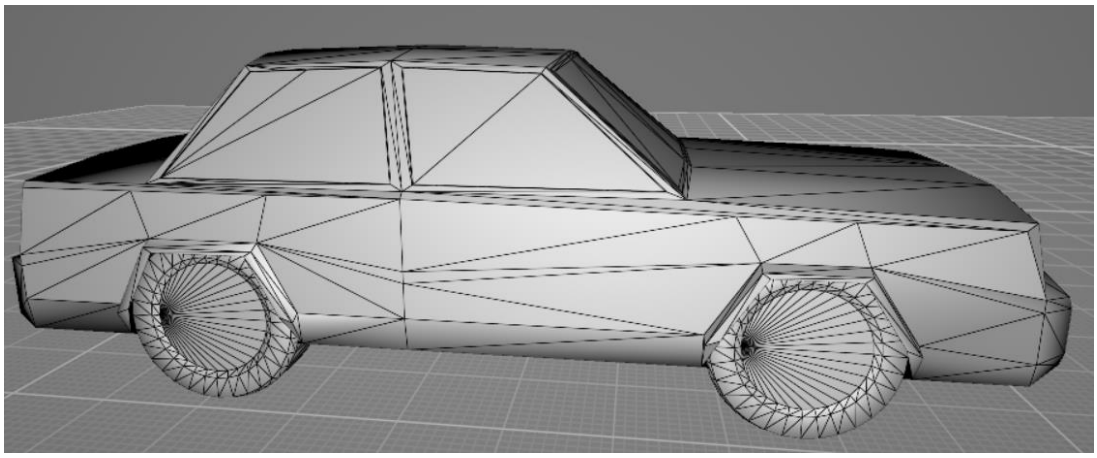


Figure 1 The above car model consists of 2,564 triangles and 7,692 after decomposition into triangles

For the hardware implementation, the vertices and faces are extracted and saved in .hex file format which can be read using Verilog and be saved in the hardware ROM.

2.1.2. Transformation Matrix

3D Transformation is done to manipulate the view of a 3D object with respect to it's original position such that all the lines, parallelism, and proportional distances remain preserved.

In this project, we have implemented following types of transformation:

i. Linear Transformation: Move the object from one coordinate to other (addition of vertex and linear displacement)
ii. Rotation Transform: Rotation matrix is used to perform a rotation in Euclidean space. Euler angles are α, β, γ, about axes x, y, z.

$$R = R_z(\gamma)\,R_y(\beta)\,R_x(\alpha) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$
$$= \begin{bmatrix} \cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma \\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta \end{bmatrix}$$

2.1.3. Projection Matrix

The 3D model is obtained in the form of ".obj" files. This file contains an array of 3 vertices (x,y,z) in each row. The 3D model can be described using polygons. The popular method of doing this is to use triangles. The 3D model is converted to traingles. Each triangle has 3 vertices. This file is a matrix multiplied by rotation vectors based on x-axis, y-axis and distance from the centre to rotate, translate, and scale the model. Thus the model matrix will be multiplied by all three transformations.

Now the model will move from model space to world space where the position and orientation of the object is to be determined. This can be understood by assuming that a camera is at the origin and through it we are viewing the object moving. Here the camera is fixed, so the rotation or revolution or movement(face(s)) of the object can be seen as being at an angle from x,y axis and the distance would give the measure of the size of the object when viewed.

Thus another matrix multiplication is done to convert the model from model space to world space.

After the matrix multiplication the image(triangle(s)) is projected onto a 2D plane and this gives the projection matrix. There are 6 main parameters: near, far, left, right, top and bottom. Near and far are used to specify the camera's field of view, that is, how near and far the camera is able to see. Left, right, top and bottom represent the boundaries of the display in their respective directions. These parameters are taken into account and multiplied with the model matrix.

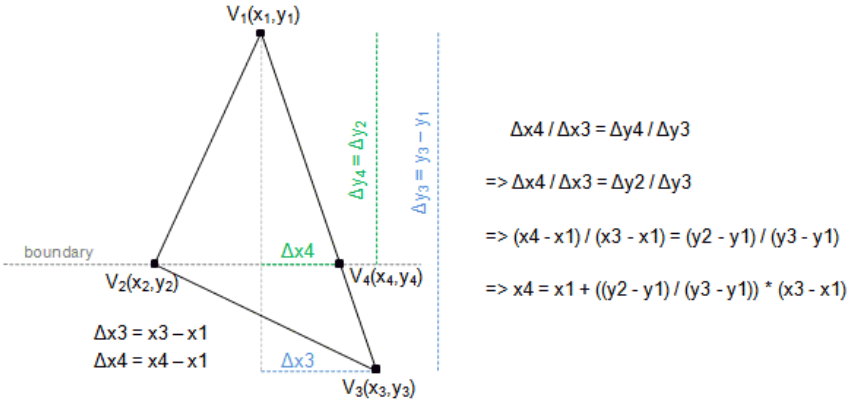
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z/f \end{bmatrix} \Rightarrow \left(f\frac{x}{z}, f\frac{y}{z} \right)$$

$$\Rightarrow (u, v)$$

2.1.4 Rasterization

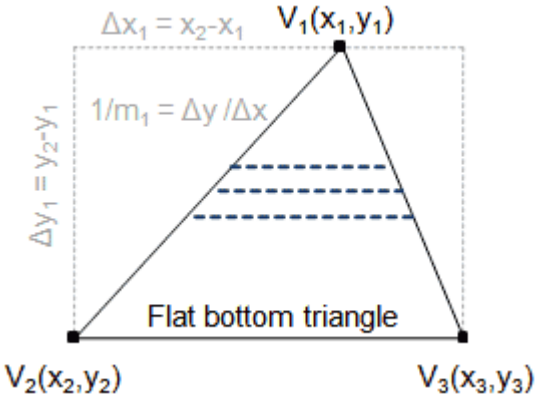
The projection matrix is now passed for rasterization. Rasterization is the process of giving colour to the pixels for display. This can be done by tracing a ray through every pixel in the image to find out the distance between the camera and any object this ray intersects (if any). The object visible through that pixel is the object with the smallest intersection distance. Another way to rasterize is to iterate for every triangle and in the iteration run through the entire block to search the pixels that fall in the triangle. There are two ways to perform this, edge detection using centroid and scan line algorithm. In this project we have adopted the scan line algorithm.

In the scan line method, the basic idea of drawing a general triangle is to decompose it into two triangles - a flat bottom triangle and a flat top triangle -

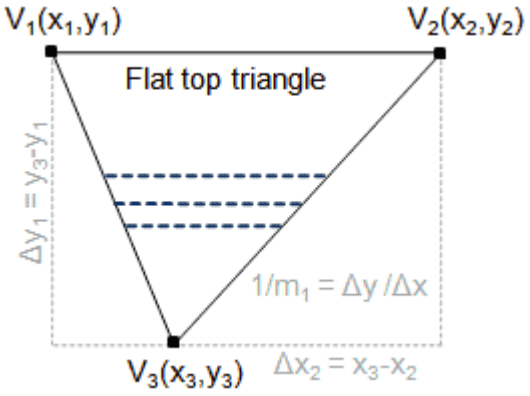


and draw them both.
The main task is to find the intersection boundary which splits the triangle. Actually we need to find a fourth vertex v4 which is the intersection point of the boundary line and the long edge of the triangle. Let's have a look at outline below:
The standard algorithm uses the fact that the two easy cases, a flat bottom and a flat top triangle, are easy to draw. The algorithm now performs as following:
First calculate inverse slope for each leg. Then we just iterate from top (y1) to bottom (y2) and add invslope1 and invslope2 to temporary variables - this gives the endpoints of a straight line.
The pseudo code for filling the bottom flat and top flat triangles is given as below:

```
fillBottomFlatTriangle(Vertice
v1, Vertice v2, Vertice v3)
{
  float invslope1 = (v2.x - v1.x)
/ (v2.y - v1.y);
  float invslope2 = (v3.x - v1.x)
/ (v3.y - v1.y);
  float curx1 = v1.x;
  float curx2 = v1.x;
  for (int scanlineY = v1.y;
scanlineY <= v2.y; scanlineY++)
  {
    drawLine((int)curx1, scanlineY, (int)curx2,
scanlineY);
    curx1 += invslope1;
    curx2 += invslope2;
  }
}
```



```
fillTopFlatTriangle(Vertice v1, Vertice v2, Vertice v3)
{
  float invslope1 = (v3.x - v1.x)
/ (v3.y - v1.y);
  float invslope2 = (v3.x - v2.x)
/ (v3.y - v2.y);
  float curx1 = v3.x;
  float curx2 = v3.x;
  for (int scanlineY = v3.y;
scanlineY > v1.y; scanlineY--)
  {
    drawLine((int)curx1, scanlineY, (int)curx2,
scanlineY);
    curx1 -= invslope1;
    curx2 -= invslope2;
  }
}
```



2.1.5. VGA Display

For display of a rendered 3D model on a screen after rasterization, we implemented a VGA driver in hardware. Driving a VGA screen requires manipulating two digital synchronization pins and three analog colour pins (RED, GREEN, and BLUE). One of the synchronization pins, HSYNC, tells the screen when to move to a new row of pixels. The other synchronization pin, VSYNC, tells the screen when to start a new frame. The protocol is described below, both textually and visually.

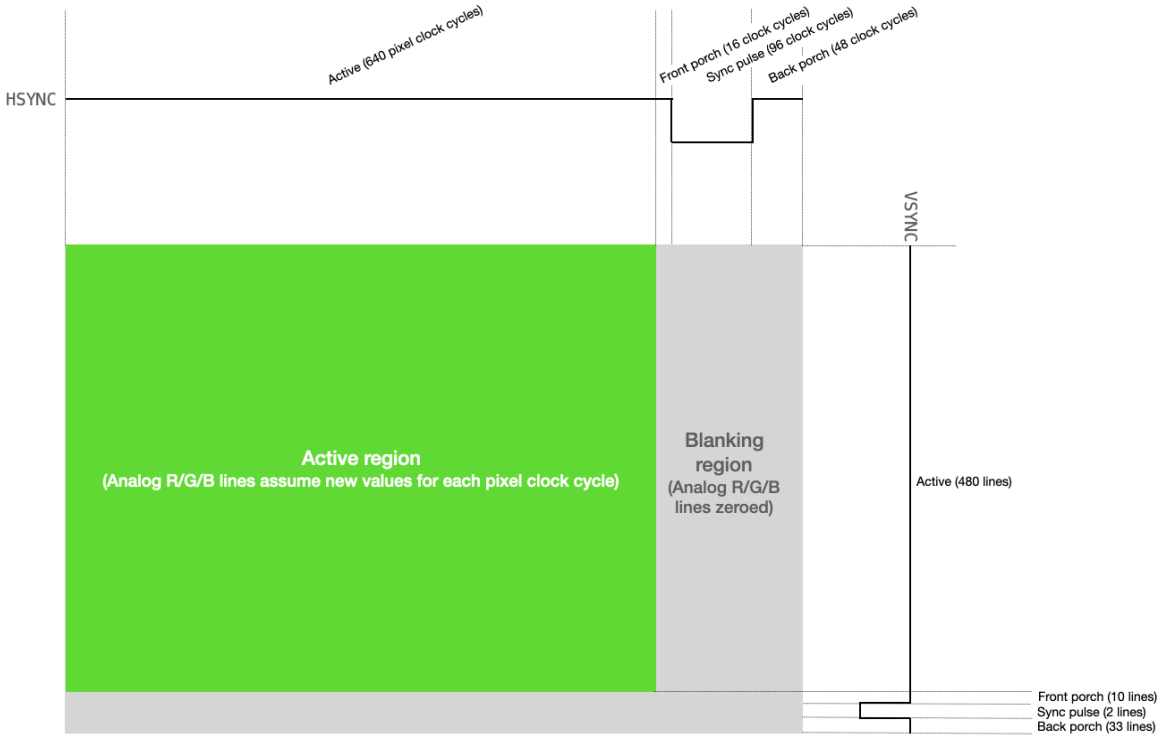


Figure 2 VGA driver description

In the hardware implementation on FPGA, we can directly generate digital signals (CLK, HSYNC, VSYNC). But for analog signal to represent RED, GREEN and BLUE color, we require a DAC (Digital to analog converter). So, we developed a simple Binary Weighted Resistor D/A Converter Circuit with 3 bits of red, 3 bits of green, and 2 bits of blue using following schematic as shown below:

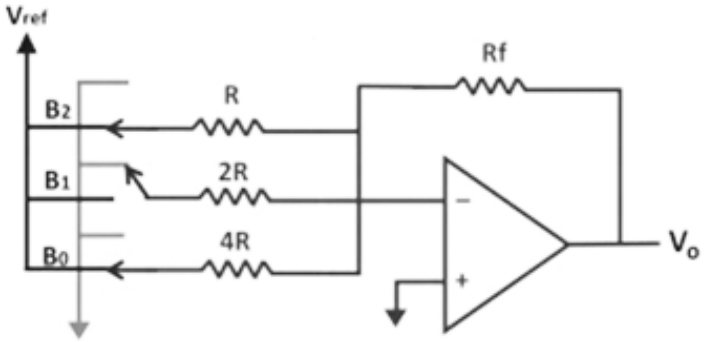


Figure 3 Circuit used for developing simple DAC

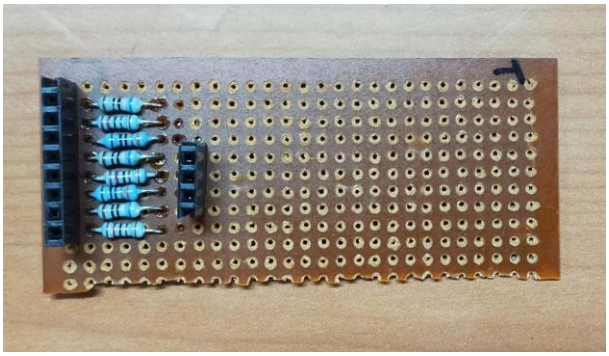


Figure 4 Binary Weighted Resistor D/A Converter Circuit

3.Software Implementation

Number Representation
We used our software prototype to confirm the types and resolution of the numbers in our
application. This tables lists the representation of the data we used throughout the hardware.

Value	Signed/Unsigned	Range	Total Bits	Integer Bits	Fractional Bits
Coordinate	Signed	[-20,20]	16	8	8
Vertex	Unsigned	[0,65535]	16	16	0
Pixel	Unsigned	[0,1]	1	1	0
Pixel color RED	Unsigned	[0, 7]	3	3	0
Pixel color GREEN	Unsigned	[0, 7]	3	3	0
Pixel color BLUE	Unsigned	[0, 3]	2	2	0

Input Ports
A main module is programmed in Verilog which does the complete 3D rendering. It takes following inputs through the module input ports:

Signal	Pins Required	Description
Clk	1	Input clock signal to the hardware

Reset	1	Resets the computation done on the original 3D object
Clr_Screen	1	Clears the VGA display
Dx_coordinate	16	User input to translate the object in x-direction
Dy_coordinate	16	User input to translate the object in y-direction
Dz_coordinate	16	User input to translate the object in z-direction
Alpha	16	User input to rotate the object in x-direction (in degrees)
Beta	16	User input to rotate the object in y-direction (in degrees)
Gamma	16	User input to rotate the object in z-direction (in degrees)

Output Ports

A main module is programmed in Verilog which does the complete 3D rendering. It provides following outputs through the module output ports used for connection to VGA enabled monitor display:

Signal	Pins Required	Description
Vga_clk	1	Output synchronization clock for the VGA enabled display
Red	3	signal for the red color
Green	3	signal for the green color
Blue	2	signal for the blue color
Hsync	1	Horizontal Sync
Vsync	1	Vertical Sync
VGA_sync_n	1	Horizontal Sync inverted
VGA_blank_n	1	Vertical Sync inveretd

LUTs (Look Up Tables) used

The implementation of 3D rendering pipeline required non-linear trigonometric functions during the rotation transformation. Following two LUTs are implemented therefore in the project:

LUT name	ROM Depth	ROM Width (bits)	Signed/Unsigned	Range	Integer Bits	Fractional Bits
Sine	64	8	Unsigned	[0,1)	0	8
Cosine	64	8	Unsigned	[0,1)	0	8

4. Simulation Results:

Comparison of rasterization results in Python versus Verilog implementation:

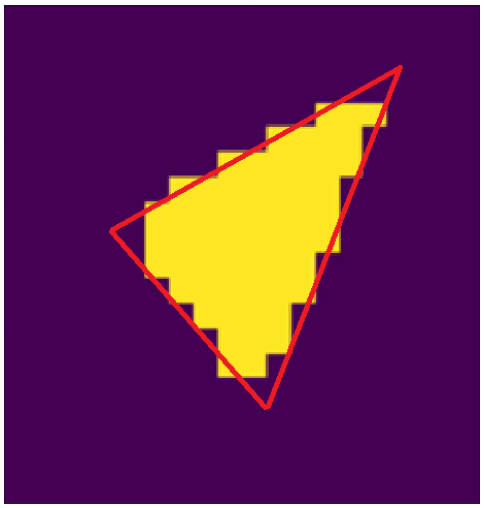


Figure 5 Rasterization of a triangle with vertices ([15, 3, 2], [5, 8, 2], [10, 15, 2]) in Python

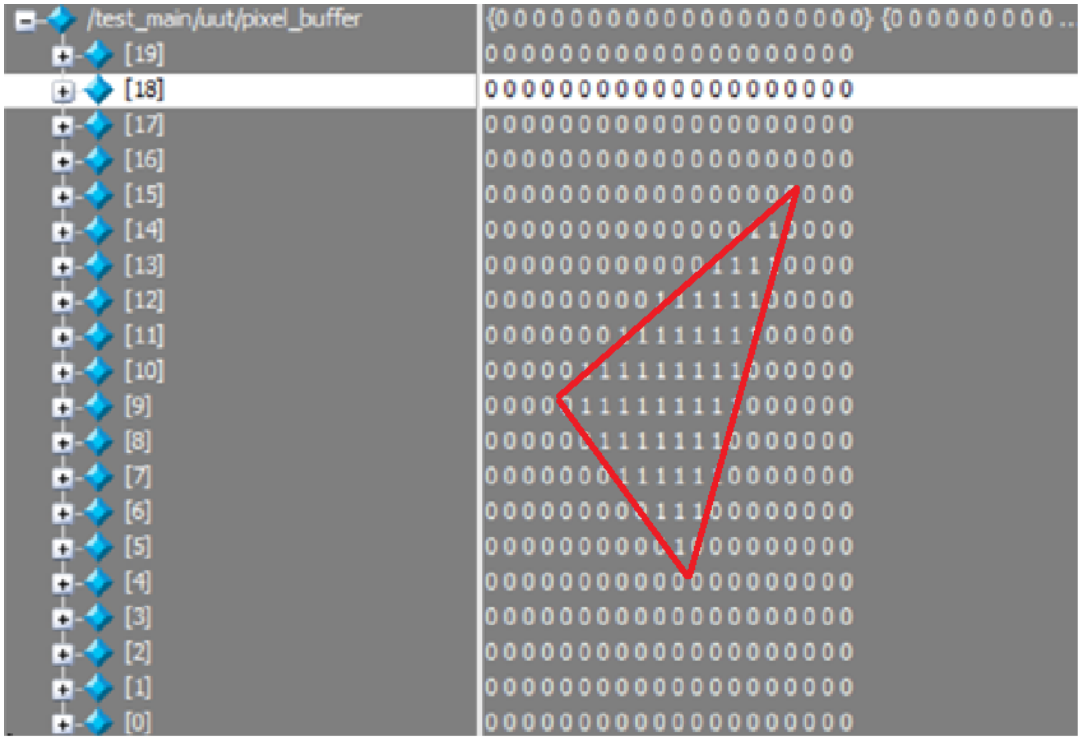


Figure 6 1-bit pixel buffer for rasterized triangle with vertices ([15, 3, 2], [5, 8, 2], [10, 15, 2]) in Verilog RTL simulation

Result of linear transformation:

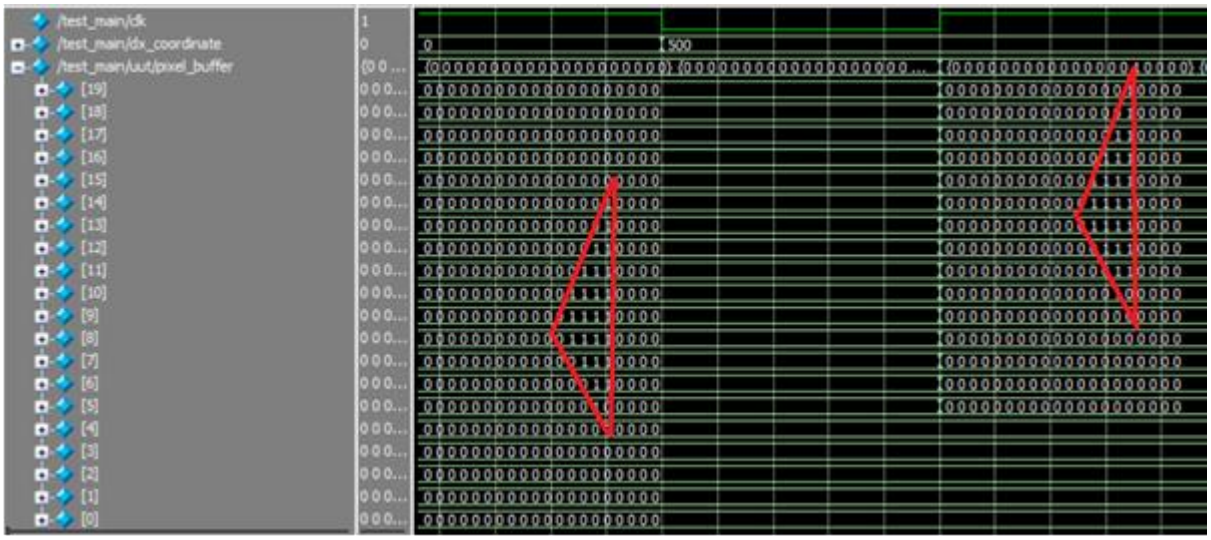


Figure 7 Transformation of triangle when it is rotated 5 units on x-axis i.e. dx_cordinate = 5.00

Result of rotational transformation:

Figure 8 Transformation of triangle when it is rotated 60 degrees on x-axis i.e. alpha = 60

VGA driver:

Linear Translation in X-axis:

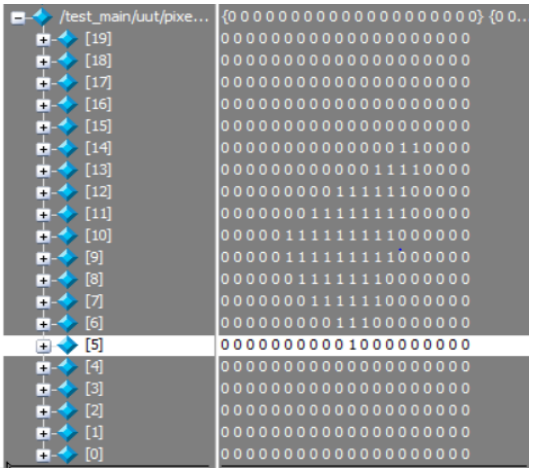


Figure 9 Rasterized Pattern Before Translation

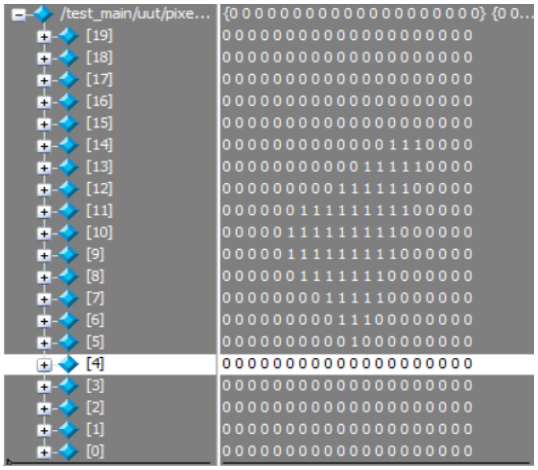


Figure 10 Rasterized Pattern After Translation

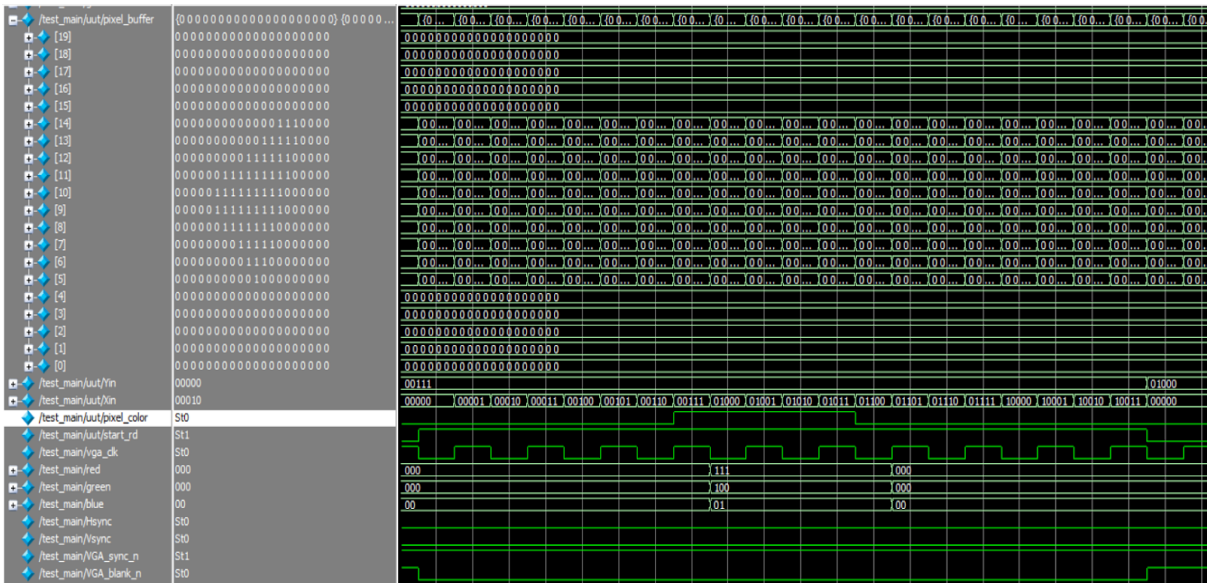


Figure 11 VGA Output for Translated Pattern

- The rasterized pattern of the original model is stored in the pixel buffer. Bit '1' shows the presence of the model on the primary screen.
- In the rasterized pattern of the linearly translated model along the x-axis is stored in the pixel buffer.
- For pixel buffer values equal to '1' the output is plotted onto a VGA screen with color configuration as red = '111' , green = '100' and blue = '01'.
- Fig shows the output to the vga display given as per the values stored in the pixel buffer.

Angular Translation:

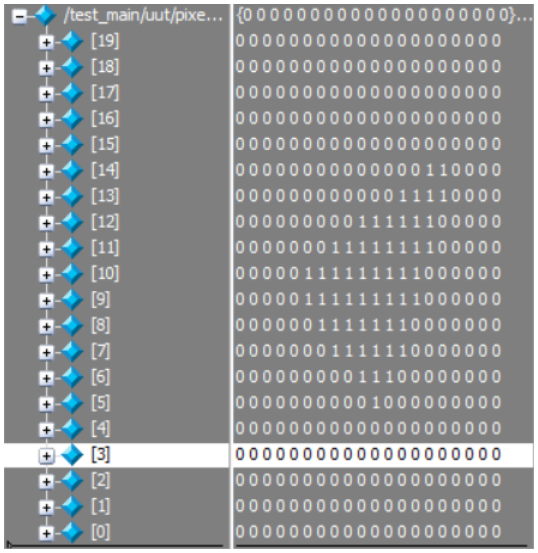


Figure 12 Rasterized Pattern Before Translation

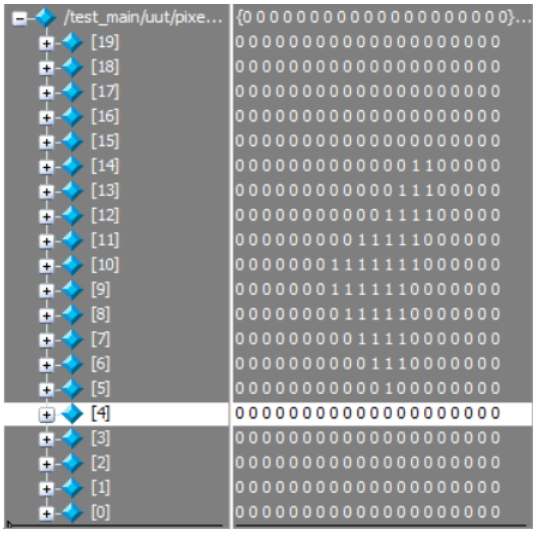


Figure 13 Rasterized Pattern After Translation

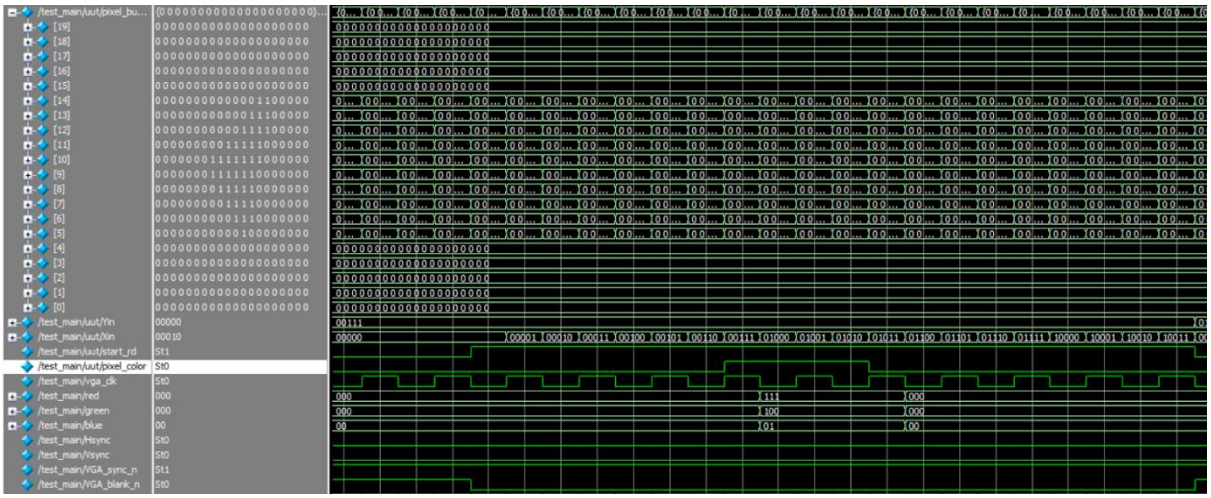


Figure 14 VGA Output for Translated Pattern

- In fig, the rasterized pattern of the original model is stored in the pixel buffer. Bit '1' shows the presence of the model on the primary screen.
- In fig, the rasterized pattern of the translated model which is translated for alpha = 30 degrees and is stored in the pixel buffer.
- For pixel buffer values equal to '1' the output is plotted onto a VGA screen with color configuration as red = '111' , green = '100' and blue = '01'.
- Fig shows the output to the vga display given as per the values stored in the pixel buffer.

RTL viewer:

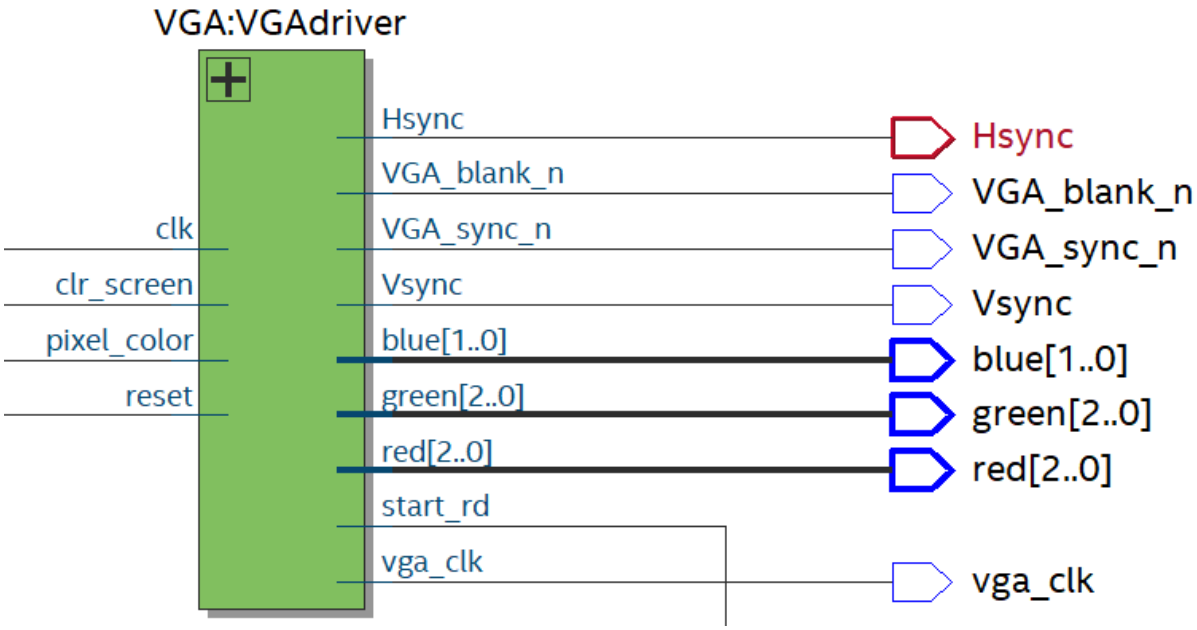


Figure 15 RTL view of VGA driver

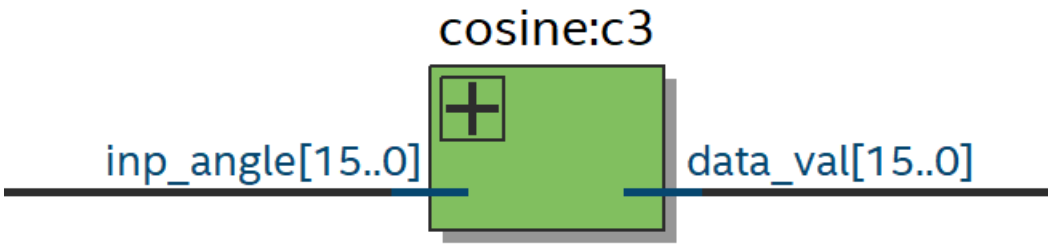


Figure 16 RTL view of Cosine LUT

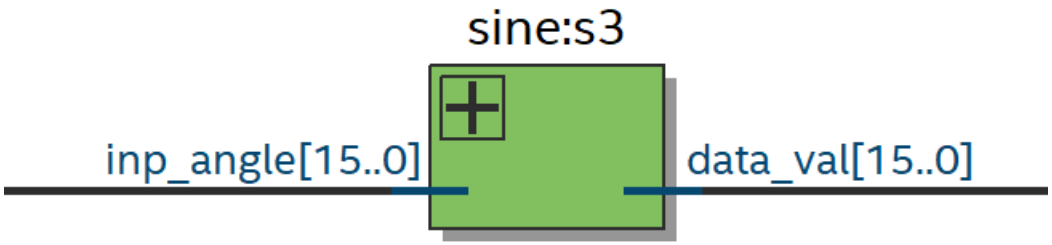


Figure 17 RTL view of Sine LUT

5. Static Timing Analysis:

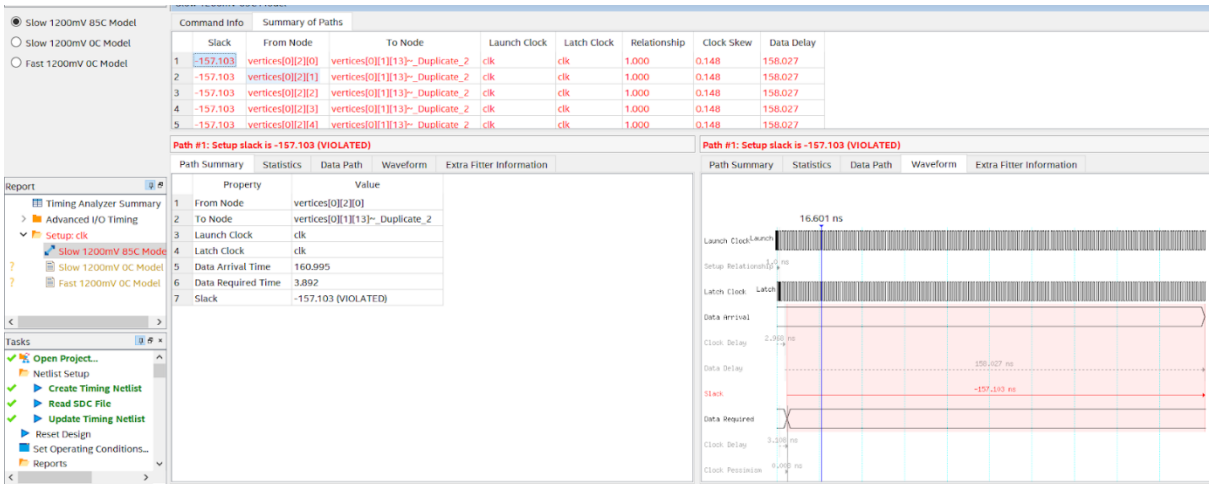


Figure 18 Static Timing Analysis for Slow 1200mV 85C model

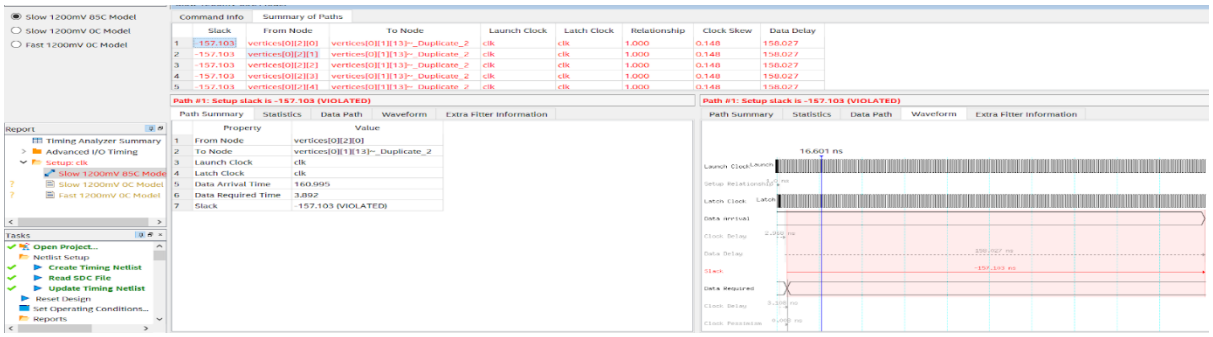


Figure 19 Path Summary for Slow 1200mV 85C Model

- Slack for default 1ns clock we get is -157.103 ns
- As slack is negative the delay is larger than the constraints.
- Worst case delay = 1ns - (-157.103 ns) = 158.103ns
- Maximum usable clock frequency, Fmax = 1/(158.103ns) = 6.32 MHz
- Clock Skew , tskew = 2.968 - 3.018 = -0.05 ns

After the Timing analysis and changing of the clock constraint to get the positive slack,

- Set clk period = 160 ns

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	1.897	vertices[0][2][0]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
2	1.897	vertices[0][2][1]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
3	1.897	vertices[0][2][2]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
4	1.897	vertices[0][2][3]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
5	1.897	vertices[0][2][4]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
6	1.897	vertices[0][2][5]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
7	1.897	vertices[0][2][6]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
8	1.897	vertices[0][2][7]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
9	1.897	vertices[0][2][8]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027
10	1.897	vertices[0][2][9]	vertices[0][1][13]~_Duplicate_2	clk	clk	160.000	0.148	158.027

Figure 20 Path Summary for the clk Period 160ns

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	vertices[0][2][0]
2	To Node	vertices[0][1][13]~_Duplicate_2
3	Launch Clock	clk
4	Latch Clock	clk
5	Data Arrival Time	160.995
6	Data Required Time	162.892
7	Slack	1.897

Figure 21 Positive Slack Obtained for clk period 160ns

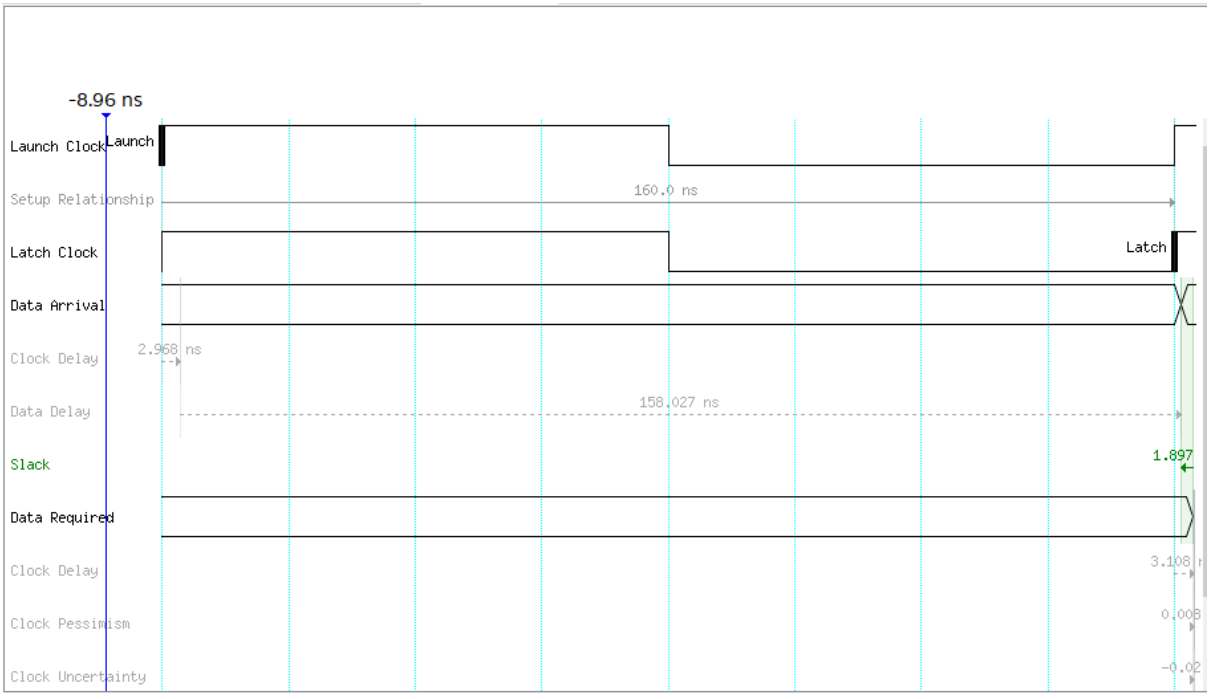


Figure 22 Timing Analysis Diagram for 160ns clk constraint

6. Layout Design:

A simple computation block which is representative of single vertex computations is developed in Verilog for the purpose of VLSI layout demonstration in Magic. This simple computation block is included in the zip file along with the magic layout for the same. A zoomed-out view of the layout is shown below for reference:

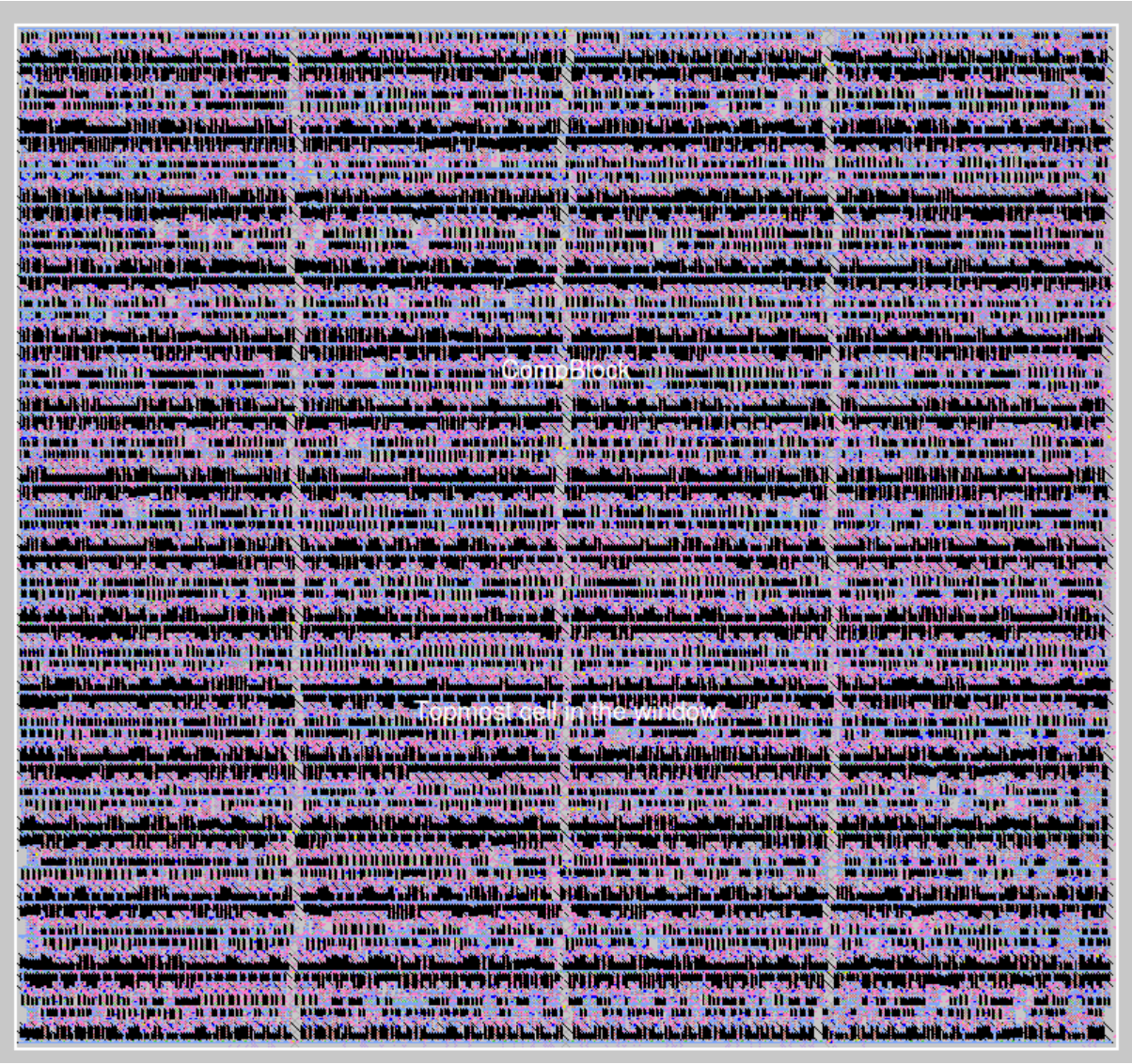


Figure 23 Layout of the simple computation block implemented in MAGIC software

7. FPGA Implementation:

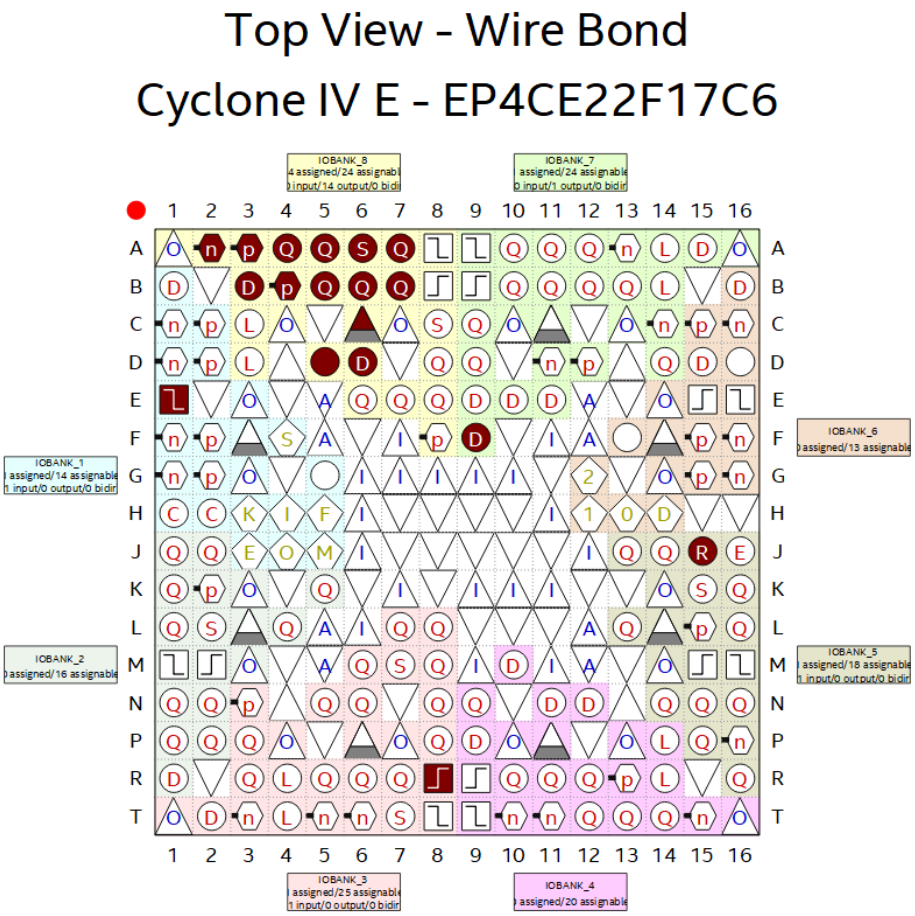


Figure 24 Pin Assignment in FPGA DE0 Nano for VGA Driver

Named: * Edit:

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Bus Hold
Hsync	Output	PIN_A3	8	B8_N0	PIN_A3	3.3-V LVTTL		8mA (default)	2 (default)		
VGA_blank_n	Output	PIN_B4	8	B8_N0	PIN_B4	3.3-V LVTTL		8mA (default)	2 (default)		
VGA_sync_n	Output	PIN_B3	8	B8_N0	PIN_B3	3.3-V LVTTL		8mA (default)	2 (default)		
Vsync	Output	PIN_A2	8	B8_N0	PIN_A2	3.3-V LVTTL		8mA (default)	2 (default)		
blue[1]	Output	PIN_B5	8	B8_N0	PIN_B5	3.3-V LVTTL		8mA (default)	2 (default)		
blue[0]	Output	PIN_A4	8	B8_N0	PIN_A4	3.3-V LVTTL		8mA (default)	2 (default)		
clk	Input	PIN_R8	3	B3_N0	PIN_R8	3.3-V LVTTL		8mA (default)			
green[2]	Output	PIN_D5	8	B8_N0	PIN_D5	3.3-V LVTTL		8mA (default)	2 (default)		
green[1]	Output	PIN_A5	8	B8_N0	PIN_A5	3.3-V LVTTL		8mA (default)	2 (default)		
green[0]	Output	PIN_B6	8	B8_N0	PIN_B6	3.3-V LVTTL		8mA (default)	2 (default)		
out_clk	Output	PIN_F9	7	B7_N0	PIN_F9	3.3-V LVTTL		8mA (default)	2 (default)		
pixel_color	Input	PIN_J15	5	B5_N0	PIN_J15	3.3-V LVTTL		8mA (default)			
red[2]	Output	PIN_A6	8	B8_N0	PIN_A6	3.3-V LVTTL		8mA (default)	2 (default)		
red[1]	Output	PIN_D6	8	B8_N0	PIN_D6	3.3-V LVTTL		8mA (default)	2 (default)		
red[0]	Output	PIN_B7	8	B8_N0	PIN_B7	3.3-V LVTTL		8mA (default)	2 (default)		
reset	Input	PIN_E1	1	B1_N0	PIN_E1	3.3-V LVTTL		8mA (default)			
start_rd_out	Output	PIN_C6	8	B8_N0	PIN_C6	3.3-V LVTTL		8mA (default)	2 (default)		
vga_clk	Output	PIN_A7	8	B8_N0	PIN_A7	3.3-V LVTTL		8mA (default)	2 (default)		
<<new node>>											

Figure 25 Pin assignment and I/O standards used according to RS-232 standards

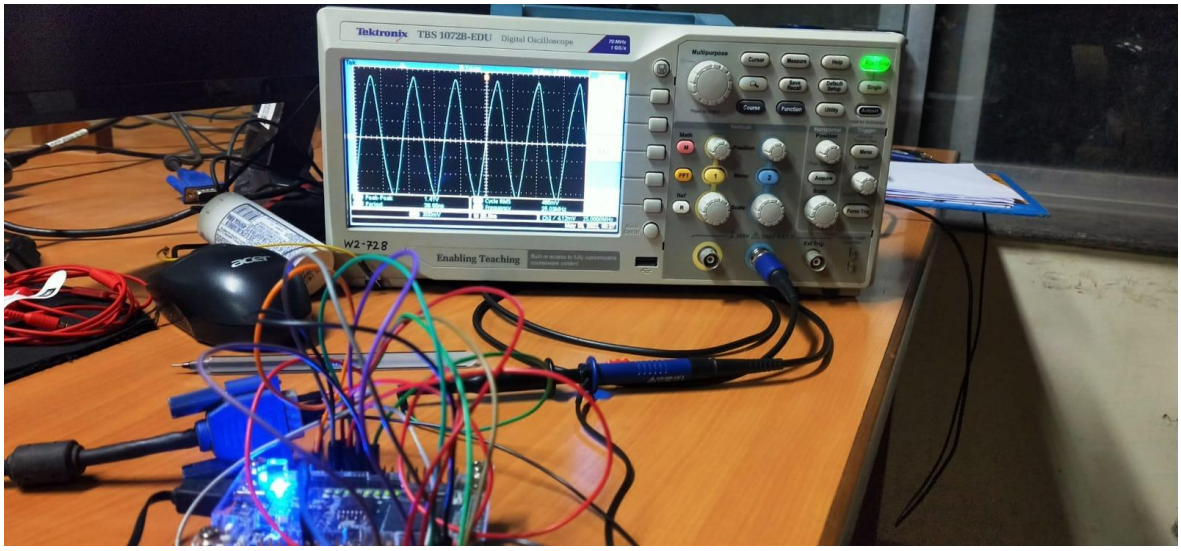


Figure 26 50Mhz Clock Test on DSO at Clk_out pin

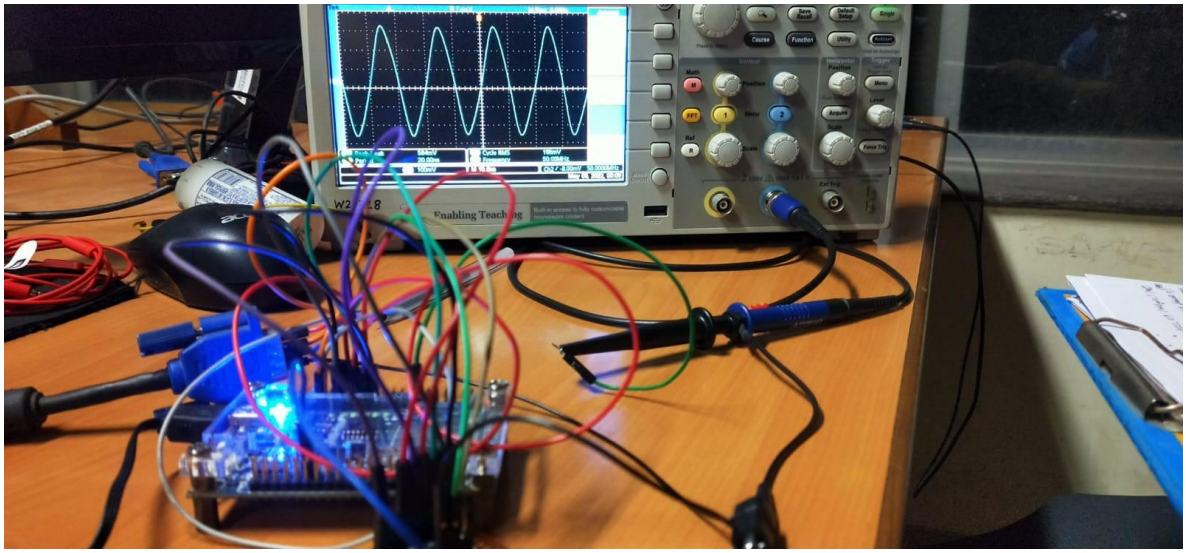


Figure 27 25MHz clock signal i.e. VGA_clk used for VGA display

8. Summary:

3D rendering is faster when implemented in hardware. In this project a small scale hardware implementation of the 3D renderer using verilog was done. A 3D model was transformed into 2D using transform matrices and projection matrix. The projection matrix is used to transform the form from 3D space to 2D space. This was followed by rasterization in which the triangles are traced and after accumulation of all, the 2D image pixels are given colour. Scan line algorithm followed by interpolation was used for implementing this. After this

VGA driver was used to output the frame buffer to the display. This process was first done in software using python. Then it was implemented using Verilog. The VGA driver code was implemented on an FPGA board (Altera DE0 nano) but it did not give any result that we wanted even though it was successfully working using simulation. A magic layout of a small part of the full code (multiplication and addition) was also done. Now, this code concept can be further extended to add shading to the final output so that it will give a realistic look. A controller can be used to change the direction of view of the object under display. This concept can be implemented on a full 3D image/video which would give insight into how gaming is done. Rendering takes a lot of time. This time can be reduced if implemented in hardware but it is a very memory hungry application.

9. **References**:

I. http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html
II. https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation
III. https://www.researchgate.net/publication/220885909_3D_rendering_using_FPGAs