Heaven's Light is Our Guide

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Rajshahi University of Engineering & Technology, Bangladesh

DESIGNING GRAPHICS PROCESSING UNIT FOR 3D RENDERING ON FPGA FOR EDUCATIONAL PURPOSE

Author

Rupak Saha

Roll: 1703134

Department of Computer Science & Engineering

Rajshahi University of Engineering & Technology

Supervised by

Nahin Ul Sadad

Assistant Professor

Department of Computer Science & Engineering

Rajshahi University of Engineering & Technology

ACKNOWLEDGEMENT

I would like to express my special appreciation and thanks to my supervisor Nahin Ul Sadad, Assistant Professor, Department of Computer Science & Engineering, Rajshahi University of Engineering & Technology. You have been a tremendous mentor for me. Again I would like to thank you for encouraging this thesis work. Your advice has been priceless. I am grateful to you for explaining the basics to me again and again. I thank you for guiding us through the whole work. I could not have done it without your continuous guidance. Thanks to all of my honourable teachers, friends & well-wishers for their guidance and motivation to complete this Thesis work. Thanks to all the people who were related to my RUET life.

13/08/2023 Rupak Saha

RUET, Rajshahi

Heaven's Light is Our Guide

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Rajshahi University of Engineering & Technology, Bangladesh

CERTIFICATE

This is to certify that this thesis report entitled "Designing Graphics Processing Unit For 3D Rendering On FPGA For Educational Purpose" submitted by Rupak Saha, Roll 1703134 in fulfilment of the requirement for the award of the degree Bachelor of Science in Computer Science & Engineering of Rajshahi University of Engineering & Technology, Bangladesh is a record of the candidates' work carried out by them under my supervision. This thesis has not been submitted for the award of any other degree.

External Examiner

Dr. Md. Nazrul Islam Mondal
Professor
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology
Rajshahi-6204

Thesis Supervisor

Nahin Ul Sadad
Assistant Professor
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology
Rajshahi-6204

Abstract

In this thesis study designing and implementation of a 3D graphics processing hardware unit using FPGA for educational purpose has been presented. The main goal was to establish a user-friendly and accessible platform that can be used as a learning tool for students and amateurs interested in computer graphics and FPGA programming. This thesis book has included hardware implementation of almost all of the basic stages of the graphics pipeline, including vertex processing, rasterization. This research has also included detailed descriptions of the underlying rendering algorithms and hardware design methods to aid comprehension. Verilog a high level hardware description language (HDL), has been utilized in Xilinx Vivado Design Suite software to demonstrate the FPGA implementation, using XC7A100T board, allowing users to comprehend the fundamental ideas of hardware design and their application in graphics rendering.

Keywords: 3D Graphics, FPGA, Hardware, Rasterization, Render

Contents

List of Figures

CHAPTER 1

INTRODUCTION

1.1 Introduction

Computer Science is a highly technical and practical field. As each individual discipline in computer science grows rapidly, their courses have become broader and specialized in scope and complexity. Since only a small portion of individual topics can be taught within short time span of undergraduate studies, these courses focus more on theoretical foundations with many higher levels of abstraction. [1]

The development of 3D graphics rendering has fundamentally changed how we engage with digital spaces, influencing everything from simulations and video games to scientific visualization and design. In three dimensional computer graphics virtual scenes are either rendered previously and then displayed or created in real time. In terms of creating scene images beforehand and then showing them as frames, is a very time consuming job. It is used in movie making though. On the other hand computer games require real time rendering as scenes change pretty frequently, thus beforehand rendering is impossible. In rendering techniques mainly two types of render approach exists, Raster and Ray Trace. In terms of Ray Tracing every Light ray emerged from a viewpoint in a scene is Traced and determined if the ray has hit any object in the scene or not. If it did hit then in the corresponding position of the screen through which the ray was passed is coloured according to the object's colour, applied shadow and reflection. Ray tracing takes a huge amount of time. So a more efficient algorithm, rasterization or scanline algorithm has been in use. This technique works by projection of 3D objects in 2D view plane. And then fills the objects structure with filled triangle or other primitives (polygons) [2].

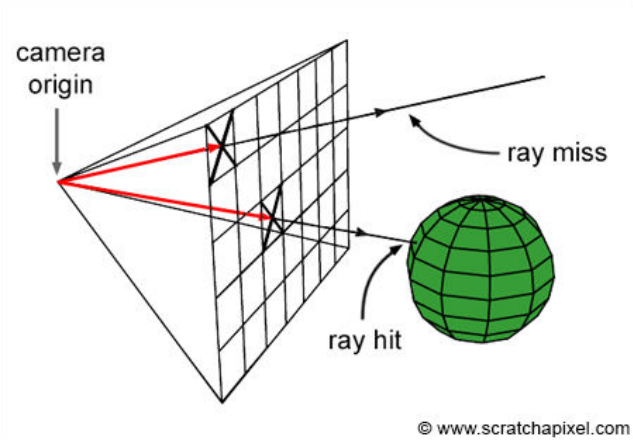

Figure 1.1: Raytrace rendering

[3]

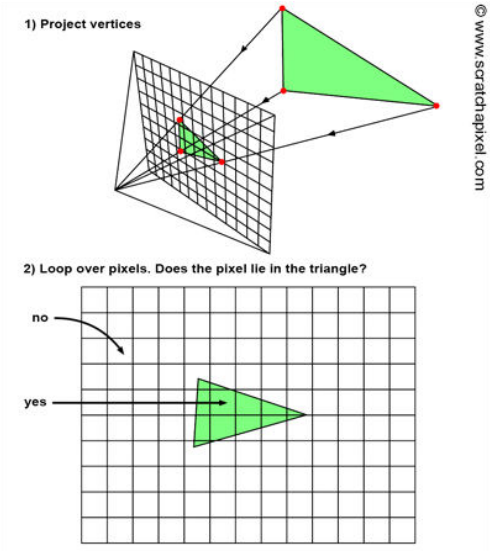

Figure 1.2: Raster rendering technique

[4]

Modern raster 3D visuals are intricate and complicated, which has sparked a growing interest in finding new rendering and accelerating techniques. Due to its inherent parallel processing powers and reconfigurability, the use of Field-Programmable Gate Arrays (FPGAs) has emerged as a potential route among these techniques. This study explores 3D graphics rendering implementation on FPGAs with a special emphasis on its usefulness in educational contexts.

1.2 Motivation

As the demand for comprehensive computer graphics education grows, students and enthusiasts seek practical ways to delve into the intricacies of rendering algorithms, pipeline stages, and hardware architectures. While software-based rendering engines offer a foundational understanding, the opportunity to engage with hardware-accelerated implementations provides a deeper insight into the interplay between algorithms and the underlying hardware. FPGAs, known for their versatility and ability to be configured into custom hardware circuits, offer an exciting platform for educators and learners alike to bridge the gap between theory and hands-on application. Thus the motivation for the work.

1.3 Literature Review

Real-time rendering performance and educational objectives may both benefit from the convergence of 3D graphics rendering and Field-Programmable Gate Arrays (FPGAs), which has become an exciting research topic. This survey of the literature seeks to give an overview of significant contributions to 3D graphics rendering on FPGAs with an emphasis on how they might be used in educational settings. Notably Authors of this Paper [5] shows how 3 dimensional models were rendered in wireframe representation using FPGA Spartan 6 device. On a single FPGA chip, the authors of this book [6] have constructed a prototype of the whole ray tracing process. Real-time frame rates of 20 to 60 frames per second are achieved while operating at just 90 MHz, and support for texturing, various light sources, and different degrees of reflection or transparency are all included. This thesis work [7] implements a two-dimensional and three-dimensional graphics processor prototype to show that a 3-D graphics processor implementation on an FPGA is viable. A completely working wireframe graphics rendering engine is created using VHDL and the development tools from Xilinx utilizing a Virtex 5 ML506 FPGA development kit.

1.4 Thesis Contribution

The main contribution of implementing a 3D graphics processing unit on a FPGA is a flexible and customizable solution for 3D rendering which helps students learn both hardware and Computer graphics rendering in depth.

1.5 Thesis Organization

The whole book is designed in such a way that it is easy to understand the flow of studies and works in a short amount of time.it includes five chapters as follows

Chapter 2 – Background Study: Theoretical Background of Computer Graphics Algorithms, description of FPGA architecture and ISE software tools used in this thesis.

Chapter 3 – Methodology: Described strategy of implementation process of Graphics processor on Xilinx Vivado Design Suite software environment.

Chapter 4 – Result Analysis: Result analysis of different modules and overall architecture.

Chapter 5 – Conclusion and Future Work: Summarization of the total work and discussion of future works.

CHAPTER 2

BACKGROUND STUDY

2.1 FPGA

FPGAs, or Field-Programmable Gate Arrays, are versatile integrated circuits that can be configured and reconfigured to perform specific tasks. These programmable devices consist of an array of logic gates and programmable interconnects, allowing engineers to design and implement custom digital circuits.

2.1.1 What Is FPGA

The customizable logic block (CLB) matrix at the centre of field programmable gate arrays (FPGAs), a semiconductor device, is coupled via programmable interconnects. FPGAs can be altered after they are created to accommodate certain application or feature requirements. This characteristic sets FPGAs apart from Application certain Integrated Circuits (ASICs), which are created specifically to meet certain design requirements. Because SRAM-based FPGAs may be updated as the design changes, they are more popular than one-time programmable (OTP) FPGAs. [8]. A variety of reconfigurable interconnects, memory blocks, and digital signal processors are among the specialized resources found inside an FPGA. Designers can determine the behaviour of the FPGA by describing the desired logic circuits in Hardware Description Languages (HDLs) like Verilog or VHDL, resulting in a hardware-software synergy. Digital signal processing, communication systems, cryptography, and most importantly the acceleration of complicated computations like graphics rendering and machine learning algorithms are just a few of the areas where FPGAs are used. FPGAs are crucial tools for building specialized, high-performance hardware solutions because of their flexibility and parallel processing capabilities, bridging the gap between software and hardware design.

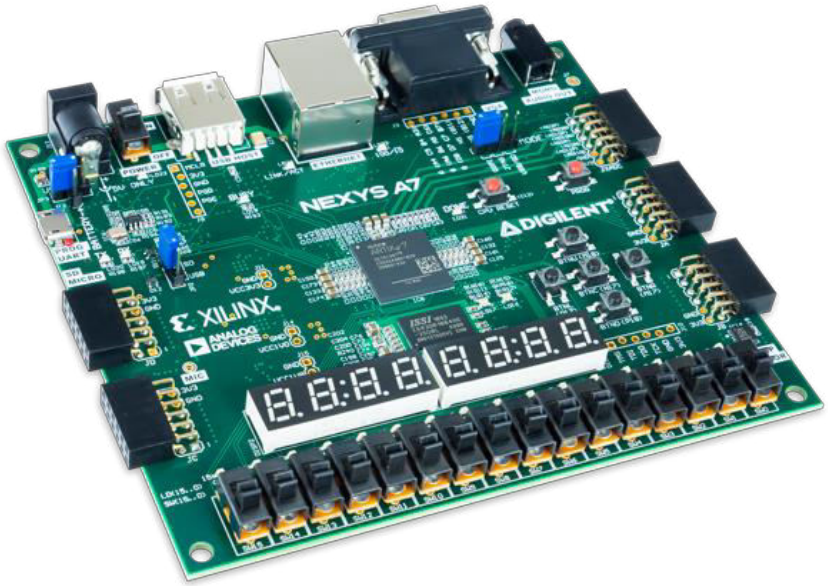

Figure 2.1: nexys7, Artix-7 FPGA

[9]

2.1.2 FPGA Architecture

An elementary FPGA design is made up of tens of thousands of essential components known as configurable logic blocks (CLBs), which are connected to one another with help of network of alterable connectors known as a fabric. The interface between the FPGA and external devices is made up of (I/O) blocks. The CLBs are also called by logic block (LB), a logic element (LE), or a logic cell (LC), depending on the manufacturer.

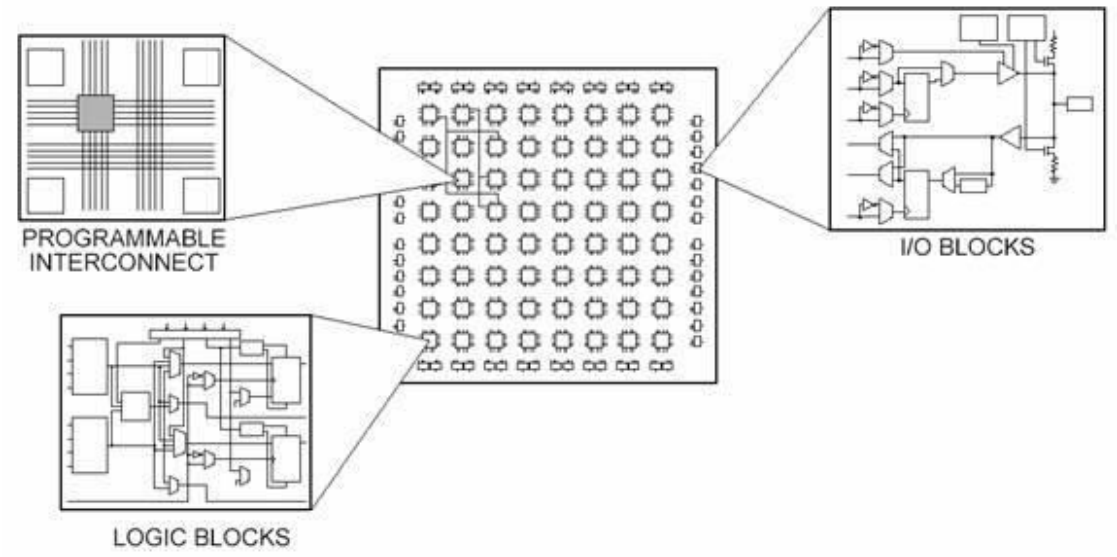

Figure 2.2: The fundamental FPGA architecture

[10]

Multiple logic blocks make compose a single CLB (Figure 2.2). An FPGA's lookup table (LUT) is one of its distinguishing characteristics. For any combination of inputs, a LUT maintains a preset list of logic outputs: It is common to utilize LUTs with four to six input bits. Multiplexers (mux), full adders (FAs), and flip-flops are typical examples of standard logic functions [11].

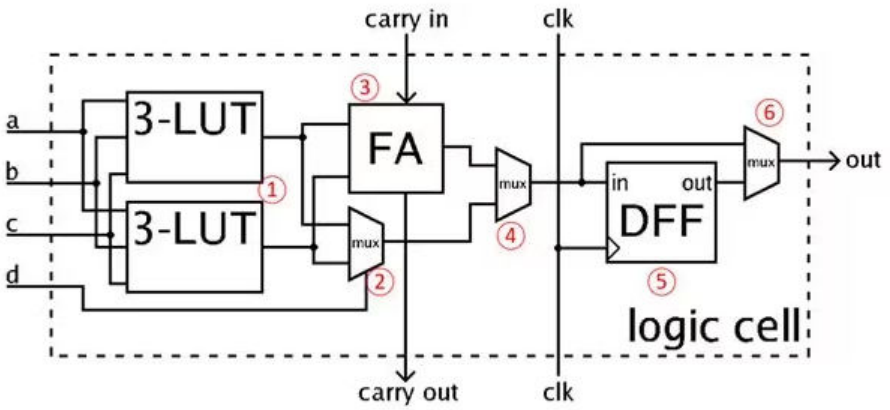

Figure 2.3: A simple CLB

[12]

The CLB's components vary depending on the device in terms of quantity and design; the simplified version in Figure 2.3, for instance, contains two three-input LUTs (1), an FA (3), a D flip-flop (5), also a regular mux (2) and two muxes which are coded into the FPGA.

The condensed CLB can work in one of two ways. The LUT outputs and a carry input from another CLB are sent as inputs to the FA in arithmetic mode. The LUTs and Mux 2 are connected in normal mode to create a four-input LUT. Mux 4 decides whether to use the FA output or the LUT output. Mux 6 determines whether an operation is going to be asynchronous or synchronized to the FPGA clock using the D FLIP-FLOP [11].

Nowadays FPGA devices includes much more complex CLBs for achieving results of complex operations.

2.1.3 Parallelism and Performance Improvement Using FPGA

The ability to use parallelism, or the reproduction of hardware processes that run concurrently in separate areas of the device, is without a doubt what sets FPGAs apart from ordinary CPUs. A straightforward algorithm and its related hardware execution are shown in Figure 2.4.

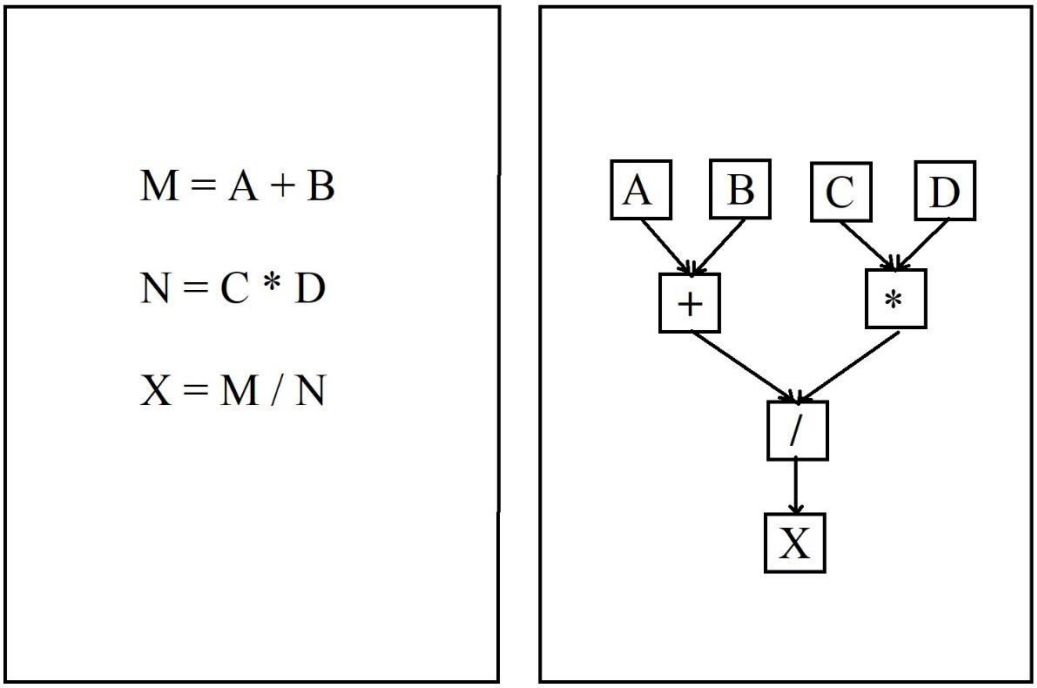

Figure 2.4: A simple algorithm and its corresponding hardware execution

Suppose two variables are A and B and result of addition on them stores in M. Another two Variables are C and D and their result of multiplication on them stores in N. If add and storing results take 4 clock cycles. And multiplication and storing takes 10 clock cycles, and division and storing result takes 12 clock cycles then a conventional CPU takes 4 + 10 + 12 = 26 clock cycles if executed sequentially. Whereas hardware execution on FPGA takes, max [4, 10] + 12 = 22 clock cycles. Thus it's visible that FPGA is a preferable over traditional CPUs in terms of parallelisation.

2.1.4 Hardware Design Scheme on FPGA

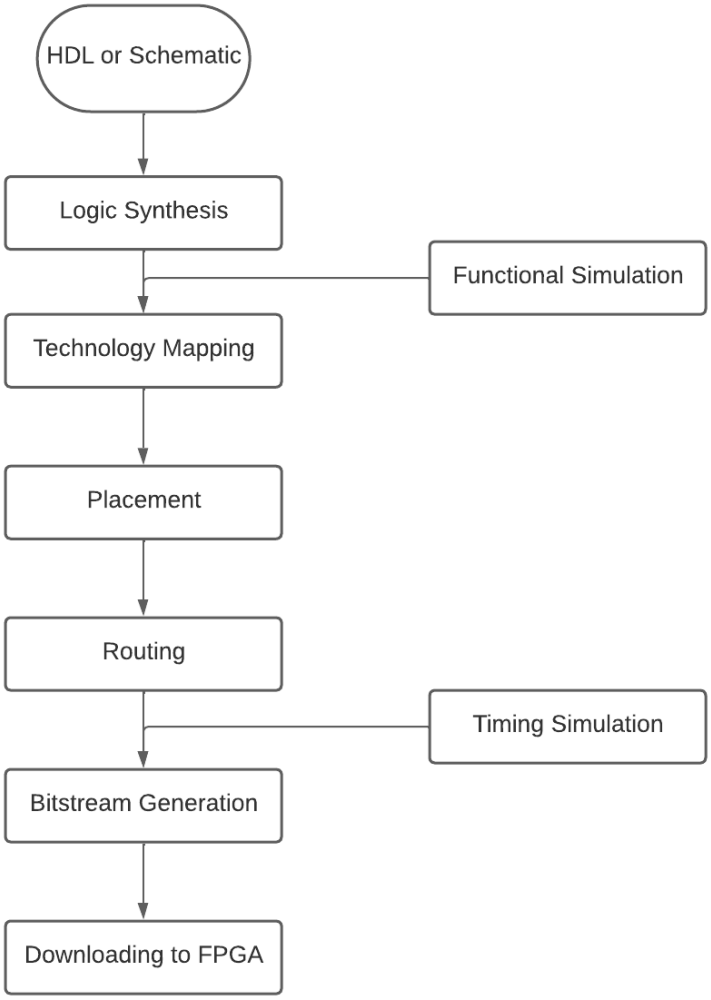

Figure 2.5: Hardware design scheme on FPGAs

The hardware realization of the method is based on the FPGA design technique shown in figure 2.5. Depending on the complexity of the design, the first step in the FPGA design process is to define the algorithm that will be implemented on the FPGA using either a schematic or a hardware description language (HDL).

The designer must verify the design's logical validity after defining it with HDLs or a schematic. Logic synthesis is used for this, followed by functional simulation. The process of logic synthesis turns an HDL or schematic-based design into a net-list of real gates and other building blocks that are defined in FPGA devices. Technology mapping is done after logic synthesis.During the synthesis procedure, also referred to as logic synthesis, the software

transforms the HDL structures into generic gate level components, such as fundamental logic gates and FFs. The implementation method is divided into three smaller steps: translate, map, and location and route. During the translation process, several design files are integrated into a single net-list. The map procedure, also known as technology mapping, maps the generic gates in the net-list to the logic cells and 10Bs of the FPGA. The place and route technique, sometimes referred to as routing and placement, is used to determine the physical layout of the FPGA chip. It establishes the actual placements of the cells and chooses the paths that various signals should take to link. The final step in the implementation process in the Xilinx flow is static timing analysis, which establishes different timing parameters, such as the maximum propagation delay and the maximum clock frequency. [13].

Make the programming file and download it. According to the final net=list, a configuration file is created in this procedure. To set the logic cells and switches, this file must be serially transferred to an FPGA device. Accordingly, the physical circuit may be confirmed [13].

2.1.5 Algorithm Implementation Strategy on FPGA

To implement algorithms on FPGAs FSM (Finite State Machine) approach is used. Finite State Machines (FSMs) play a crucial role in introducing the necessary sequentiality within algorithms, ensuring precise control and coordination of operations. In many algorithms, particularly those involving complex decision-making and conditional behaviour, the sequential execution of steps is essential for accurate results. By incorporating FSMs, designers can model and manage the distinct states and transitions within the algorithm's flow, enabling a step-by-step progression through different stages.

FSMs offer a well-organized framework for enforcing proper workflow, synchronizing data flow, and controlling communication between various parts. In situations where real-time processing, synchronization, or coordination between concurrent processes are necessary, this sequentiality is very important. When algorithms are implemented in FSMs on hardware platforms like FPGAs, the inherent parallelism of the hardware may be taken advantage of while still ensuring that complex operations are carried out precisely and deterministically. In this way, FSMs improve the general correctness and stability of the system's behaviour in addition to helping algorithms run more efficiently.

There are primarily two types of FSMs: Mealy and Moore. Mealy FSMs produce outputs based on both current states and inputs, offering flexibility but potentially leading to more complex designs. Moore FSMs, on the other hand, generate outputs solely based on the current state, simplifying the design but possibly requiring more states to achieve the same functionality.

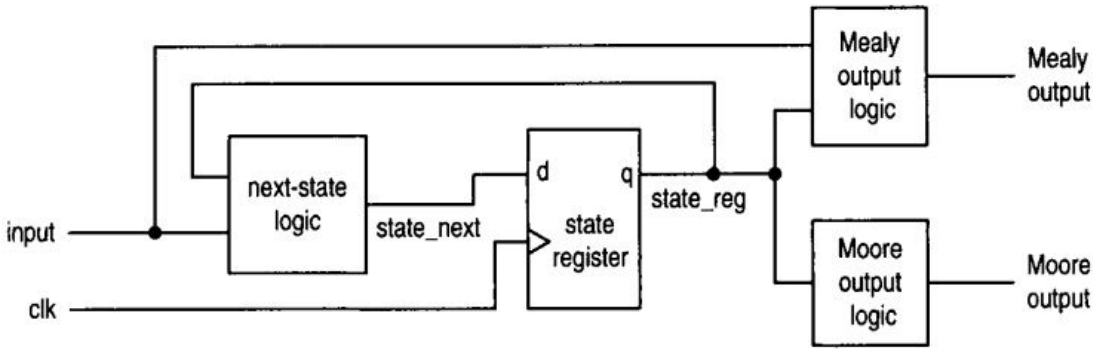

Figure 2.6: Block Diagram of a synchronous FSM containing Mealy and Moore output [13]

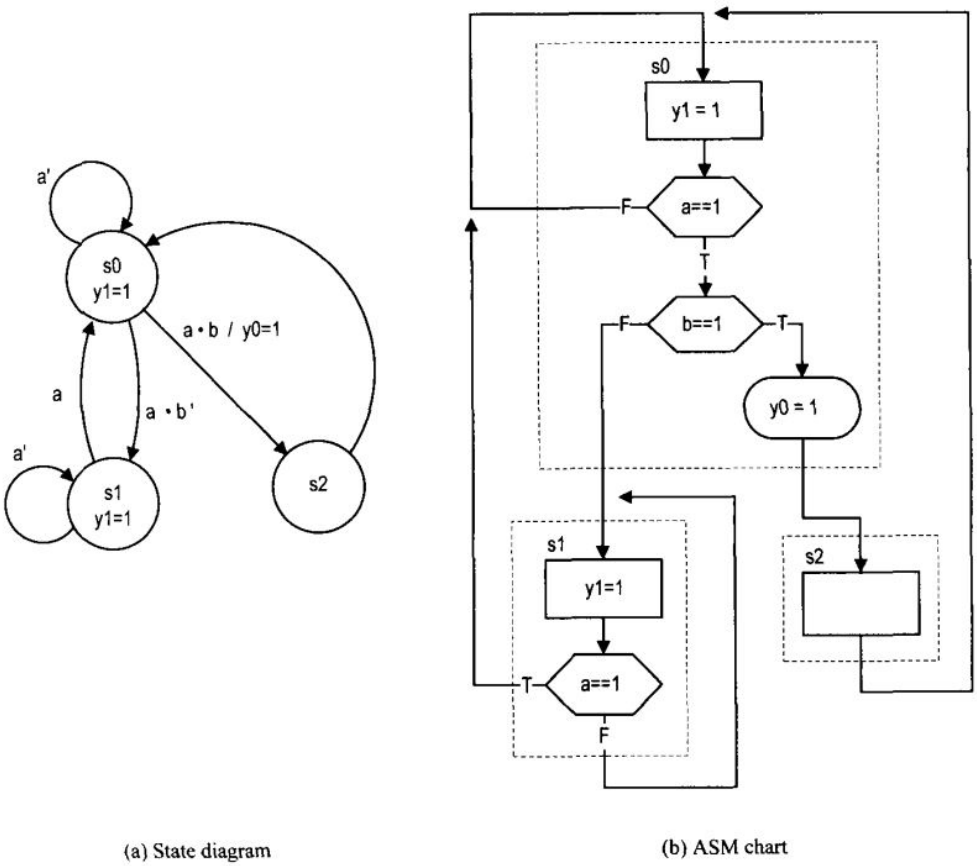

Figure 2.7: Example of an FSM

[13]

A FSM is offered in Figure 2.7 as an illustration. Two external input signals (a and b), one Moore output signal (yl), and one Mealy output signal (y0) are all present in the FSM's three states. When the FSM is in the s0 or sl state, the yl signal is asserted. When the a and b signals are "11" and the FSM is in the s0 state, the y0 signal is asserted. [13].

In Verilog, states in a Finite State Machine (FSM) are typically represented using a set of encoding strategies. These strategies determine how states are assigned to binary values, making it possible to transition between states
based on input conditions. There are different encoding strategies available, each with its own advantages and considerations:

- **Binary Encoding:** This is the most straightforward approach, where states are assigned binary values in increasing order (e.g., 00, 01, 10, 11). Binary encoding is simple and easy to understand, but it can lead to inefficient resource utilization in larger FSMs due to the increasing number of bits required for state representation [14].

- **One-Hot Encoding:** In this strategy, each state is represented by a single bit with a value of '1', while all other bits are '0'. This approach ensures that only one state bit changes during a transition, making it easy to detect state changes and reducing the risk of glitches. However, it requires more bits for encoding, which might be a concern in terms of resource usage [14].

- **Gray Code Encoding:** Gray code is an encoding scheme where adjacent states differ by only one bit. This helps to mitigate glitches during state transitions and provides a balanced approach between binary and one-hot encodings. Gray code encoding can be beneficial for FSMs with a moderate number of states [14].

- **Custom Encoding:** Depending on the specific requirements of the design and the FPGA's architecture, designers might opt for custom encodings that optimize for factors such as timing, area, or power consumption. Custom encodings involve a trade-off analysis to determine the best representation for a particular application.

The choice of encoding strategy depends on the design's complexity, timing constraints, resource availability, and desired optimization goals. Different strategies offer varying trade-offs between factors like design simplicity, resource utilization, and transition stability, allowing designers to tailor the encoding approach to the specific needs of their project.

2.1.6 Project Implementation strategy using Vivado Design Suite software

Steps: The following steps are normally taken to implement any algorithm in FPGA devices

- Create a new Project
- Specifying project type
- Set project Properties
- Add design sources
- Add constraints
- RTL analysis
- Logic Synthesis
- Implementation
- Bit-stream Generation
- Download to FPGA

For example the following images figure 2.8, figure 2.9, figure 2.10 & figure 2.11 are the screen capture of different steps taken in development of this GPU in FPGA.

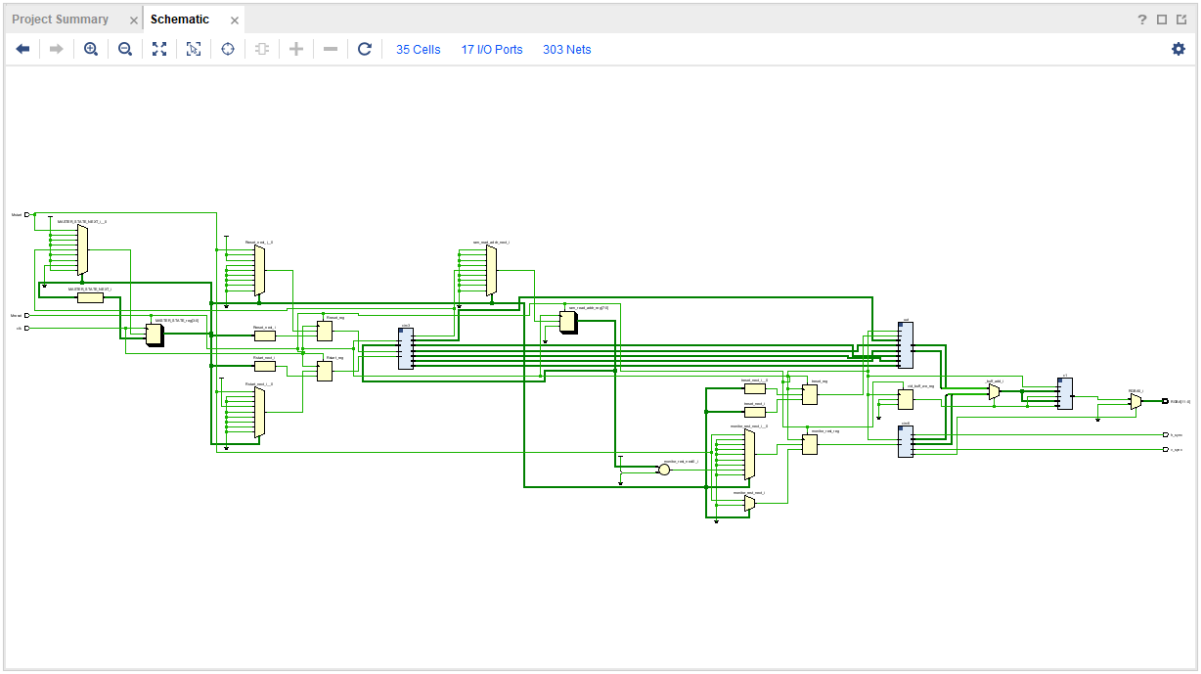

Figure 2.8: Elaborated Design (schematic)

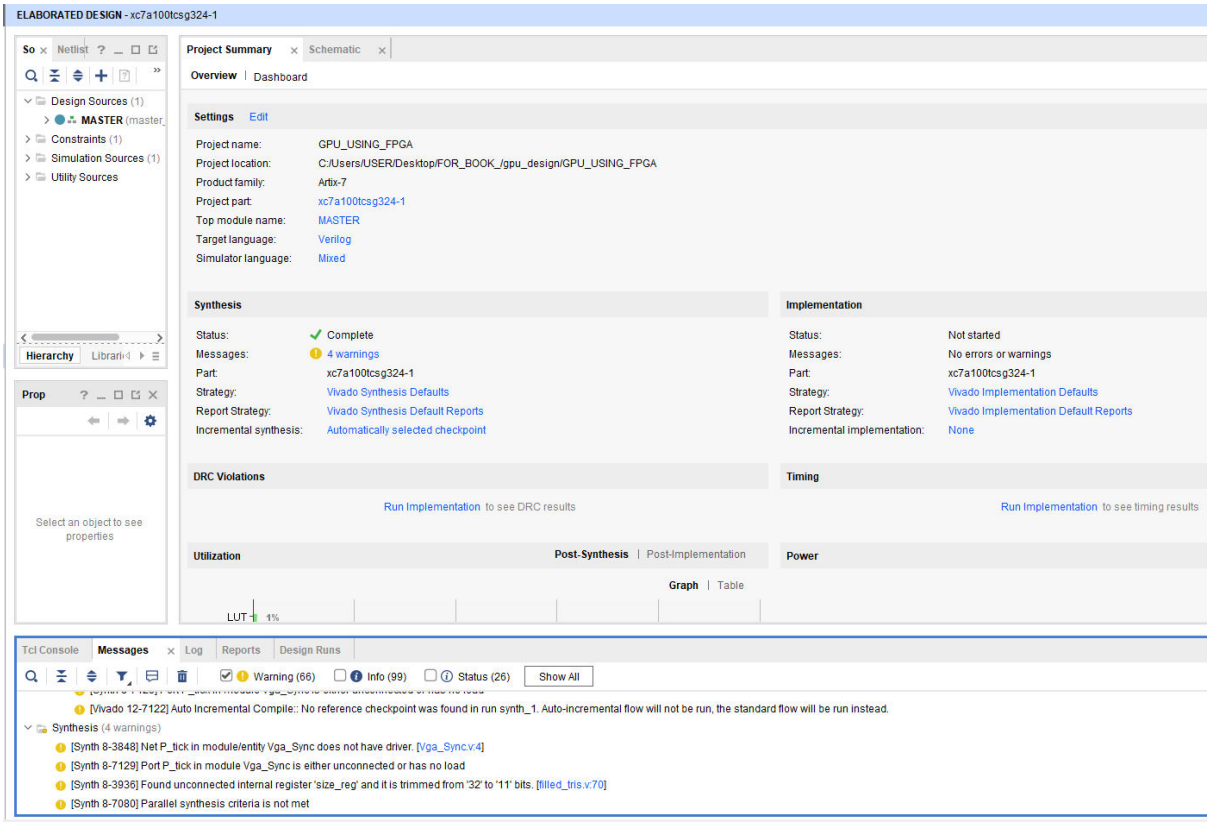

Figure 2.9: Synthesis

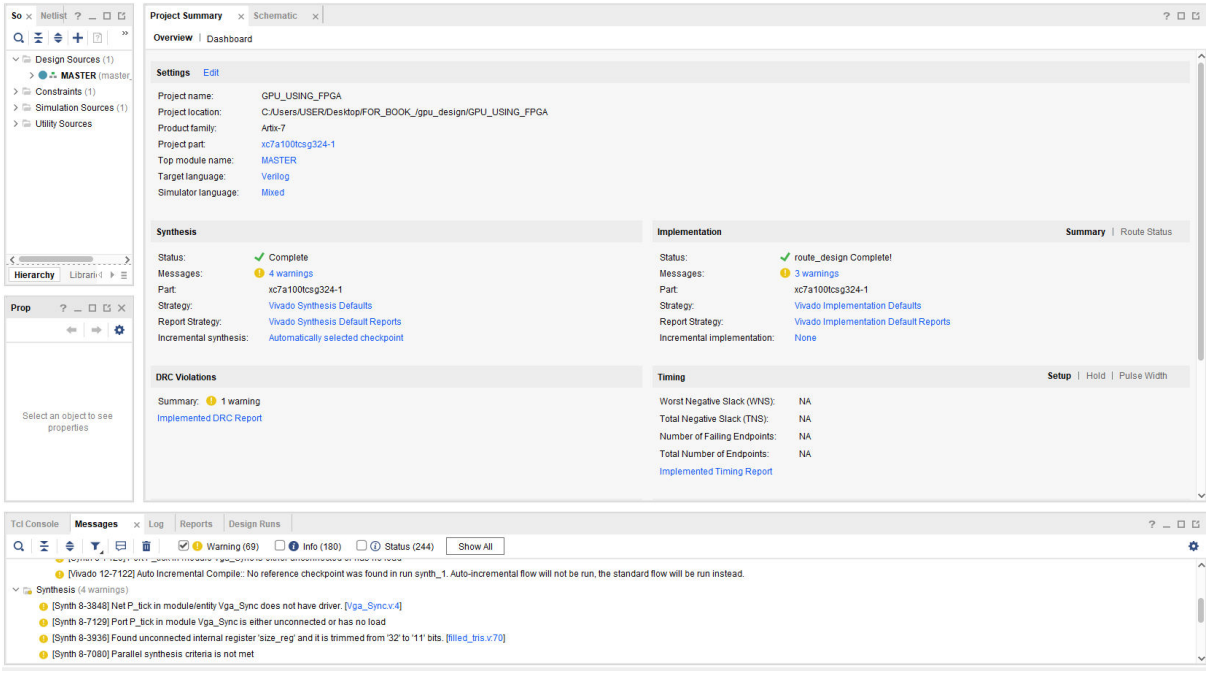

Figure 2.10: Implementation

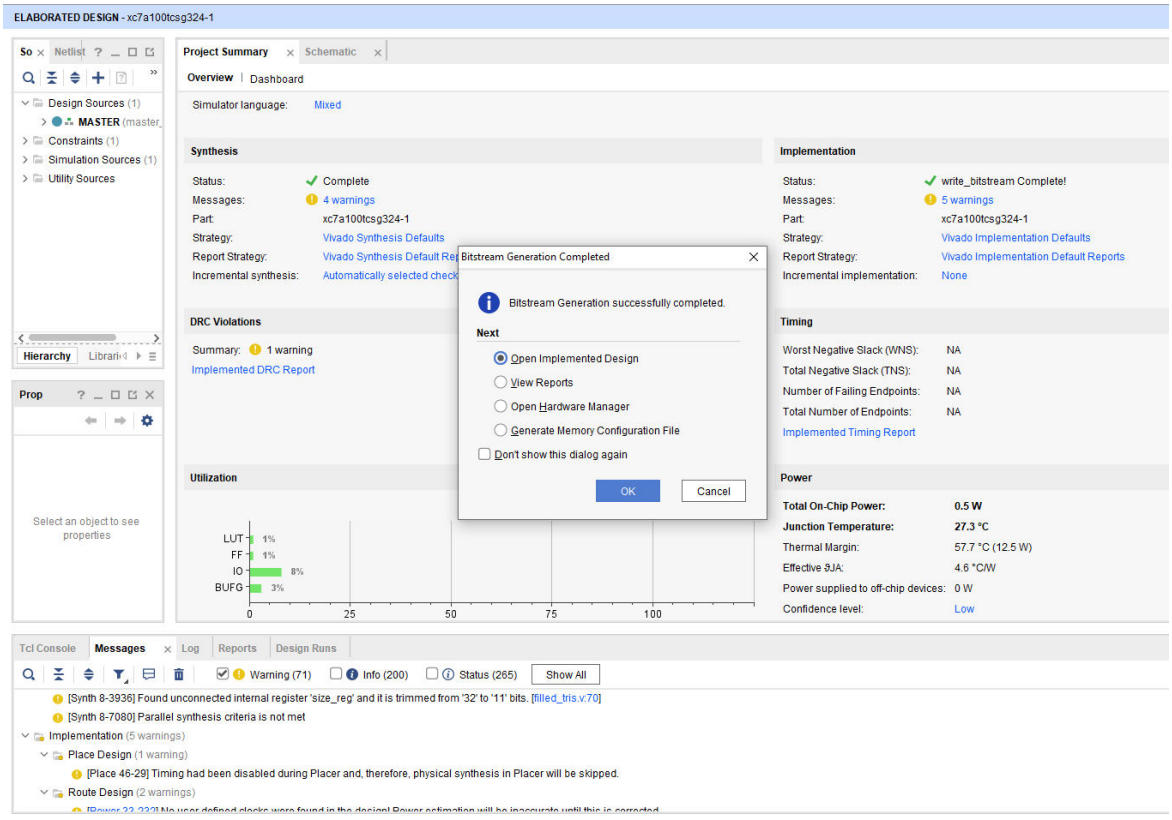

Figure 2.11: Bit stream generation

2.2 Graphics Rendering

The technique of showing a picture on a computer screen from a list of geometrically specified objects is known as computer graphics rendering. Both two-dimensional and three-dimensional spaces can be used to define these items. A virtual world is the name given to this 2D or 3D environment. Despite the three-dimensional nature of descriptions from virtual worlds, computer screens have only two dimensions. A 3 dimensional picture must be shown on a 2 dimensional plane. a two-dimensional projection of the three-dimensional image is required for computer displays.

The majority of computer screens today are raster-based. Raster graphics are a rectangular grid of colour spots (known as pixels) that are used to depict a picture. Two-dimensional vector-formatted projected objects must be quantized and transformed to raster images. This is referred to as rasterization [7].

2.2.1 Homogenous Vectors

2D and 3D objects in computer graphics are defined in relation to a virtual world. This virtual universe is nothing more than a mathematical simulation of the geometry of the real world. Objects can be defined in a variety of ways, but the most common is as a collection of vertices. As seen in Equation 2.1, these vertices are represented as vectors.

$$2D = \begin{bmatrix} x \\ y \end{bmatrix} \qquad 3D = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Equation 2.1: 2D and 3D vectors

The x, y, and z values of each vector represent an object's location within a coordinate system. Section 2.2.2 discusses coordinate systems. In computer graphics, vertices are frequently given in homogeneous form. Homogenous representation allows affine geometric transformations to be simply expressed by a single matrix multiplication. The coordinates are the same as previously in this form, except that an additional w coordinate is added to the vector as illustrated in Equation 2.2.

$$2D = \begin{bmatrix} x \\ y \\ w \end{bmatrix} \qquad 3D = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Equation 2.2: 2D and 3D homogenous vectors

2.2.2 Coordinate System

In a two-dimensional plane, we can choose any point to serve as a reference point known as the origin. We build two perpendicular number lines called axes across the origin. These are commonly referred to as the x and y axes (see figure 2.8).

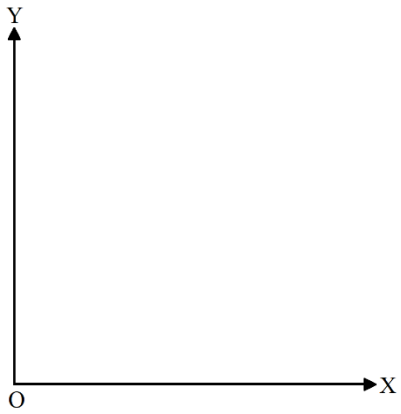

Figure 2.12: 2D coordinate system

The 3D Cartesian coordinate system is made up with an origin and three mutually perpendicular lines that pass through it. These mutually perpendicular lines are considered number lines and are labelled as the x, y, and z coordinate axes. The labels are positioned on the axes' positive ends (see Figure2.9).

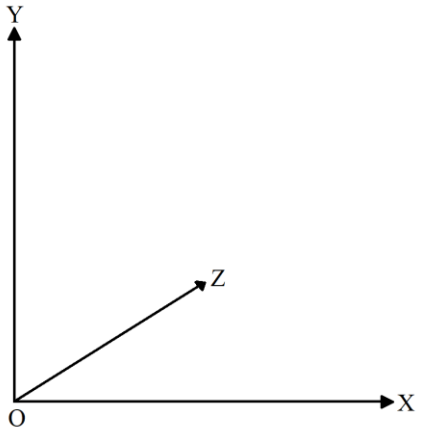

Figure 2.13: 3D coordinate system

2.2.3Model Representation

To represent 2D or 3D objects, most commercial graphics processing units nowadays employ a polygon mesh. A polygon mesh is a collection of vertices that are linked together to form an object's polygons. In this situation, the polygon primitive is a triangle. Figure 2.10 illustrates a four-point object defined as a polygon mesh. V1, V2, V3, and V4 are the vertices that comprise an object, whereas P1 and P2 are the primitive polygons specified by this set of vertices and colours.

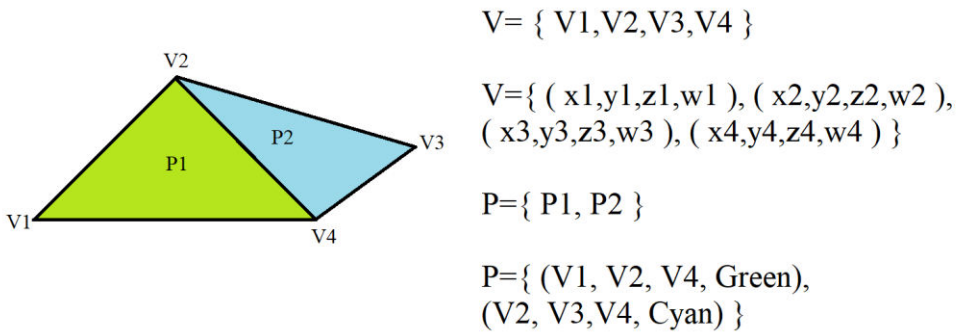

Figure 2.14: Polygon mesh representation

The list of vertices in Figure 2.7, such as V1, V2, V3, and V4, may be simply transformed from one coordinate system to another using a linear transformation matrix. The next section explains how geometric transformations can be used to scale, rotate, or translate these objects.

2.2.4 Geometric Transformations

- **Scale:** Within a coordinate system, scaling extends or compresses vertices. Each vector V is multiplied by the scaling coefficients Sx, Sy, and Sz. Below are the algebraic and matrix representations (for 2D, delete all z components).

$$v' = S(S_x, S_y, S_z).v$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{S_x} & 0 & 0 & 0 \\ 0 & \boldsymbol{S_y} & 0 & 0 \\ 0 & 0 & \boldsymbol{S_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x.\boldsymbol{S_x} \\ y' = y.\boldsymbol{S_y} \\ z' = z.\boldsymbol{S_z} \\ 1 \end{bmatrix}$$

Equation 2.3: Scale operation

S is the scaling matrix and based on its values the vector V can be either expanded or compressed.

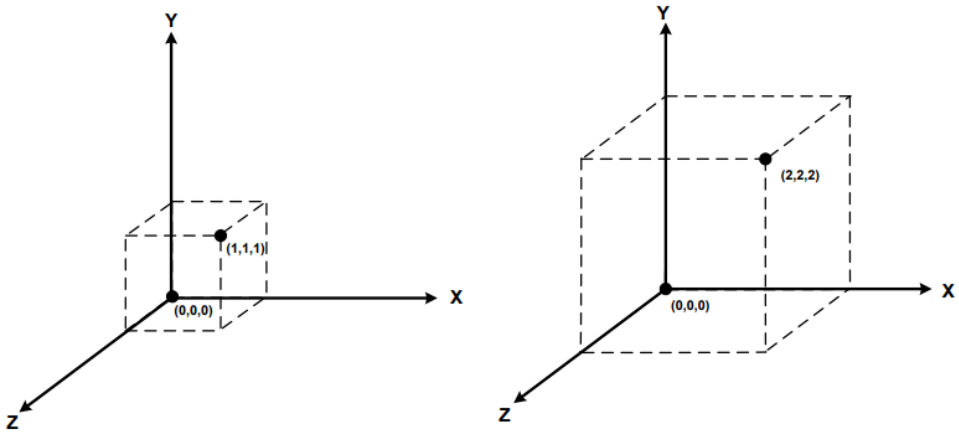

Figure 2.15: Scaling a mesh

[7]

- **Rotation:** Rotation is revolving a vector around the origin of a coordinate system. Each vector is multiplied by a rotation matrix to achieve this. A mesh in three dimensions may be rotated at three distinct angles. Each angle component is depicted in Figure 2.12 below. The following three transformation matrices can be used to rotate an object around the x, y, or z axes. These equations are shown below. R is the rotation matrix in each example, and V is the vertex being rotated

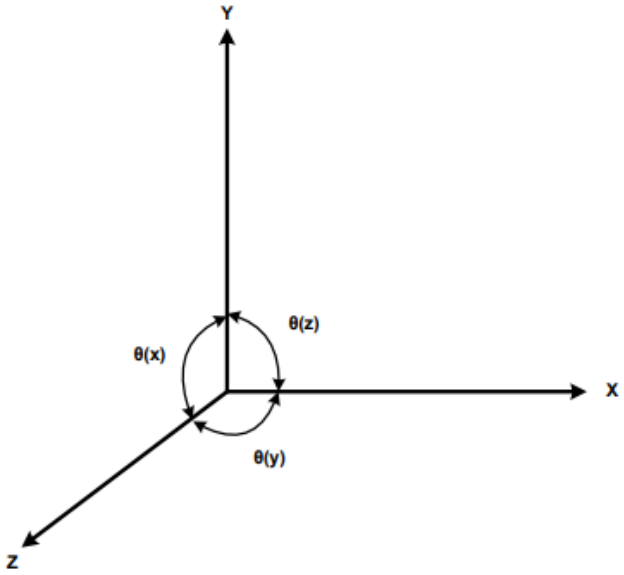

Figure 2.16: Right handed coordinate system with rotational angle

[7]

Rotation with respect to Z axis:

$$v' = R(\theta_z).v$$

$$\begin{bmatrix} \boldsymbol{x'} \\ \boldsymbol{y'} \\ \boldsymbol{z'} \\ \boldsymbol{1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{cos(\theta_z)} & \boldsymbol{-sin(\theta_z)} & 0 & 0 \\ \boldsymbol{sin(\theta_z)} & \boldsymbol{cos(\theta_z)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{x' = xcos(\theta_z) - ysin(\theta_z)} \\ y' = \boldsymbol{x\,sin(\theta_z) + ycos(\theta_z)} \\ z' = z. \\ 1 \end{bmatrix}$$

Equation 2.3: Rotation equation around Z axis

Rotation with respect to Y axis:

$$v' = R(\theta_y).v$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{cos(\theta_y)} & 0 & \boldsymbol{sin(\theta_y)} & 0 \\ 0 & 1 & 0 & 0 \\ \boldsymbol{-sin(\theta_y)} & 0 & \boldsymbol{cos(\theta_y)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{x' = xcos(\theta_y) + zsin(\theta_y)} \\ y' = y \\ \boldsymbol{z' = x\,sin(\theta_y) - zcos(\theta_y)} \\ 1 \end{bmatrix}$$

Equation 2.4: Rotation equation around Y axis

Rotation with respect to X axis:

$$\boldsymbol{v'} = \boldsymbol{R(\theta_x).v}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \boldsymbol{cos(\theta_x)} & \boldsymbol{-sin(\theta_x)} & 0 \\ 0 & \boldsymbol{sin(\theta_x)} & \boldsymbol{cos(\theta_x)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x \\ y' = \boldsymbol{ycos(\theta_x) - zsin(\theta_x)} \\ z' = \boldsymbol{y\,sin(\theta_x) + zcos(\theta_x)} \\ 1 \end{bmatrix}$$

Equation 2.5: Rotation equation around X axis

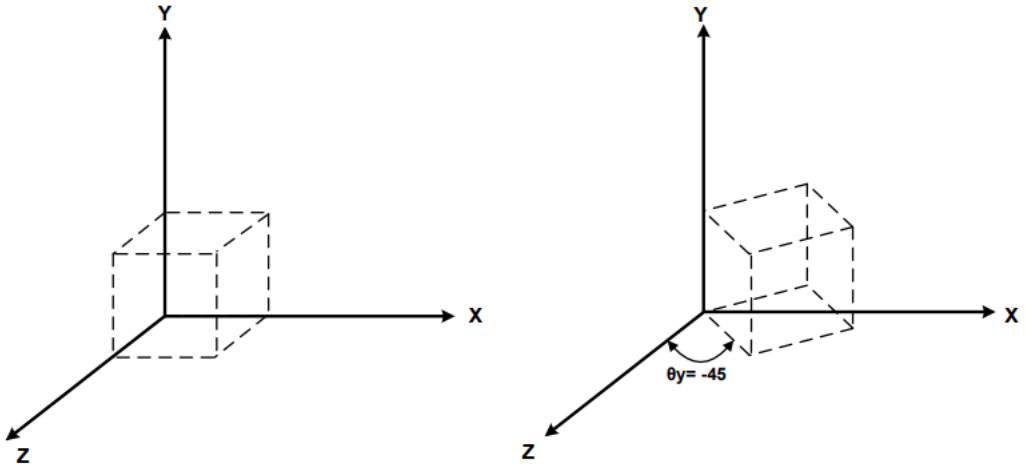

Figure 2.17: Rotation of a mesh

[7]

- **Translation:** Translation is the process of transferring vertices in a coordinate system from one position to another by adding a translation amount to each vertex. Equation below shows the algebraic and matrix representations for 3D translation (for 2D, eliminates all z components).

$$\boldsymbol{v'} = \boldsymbol{D}(\boldsymbol{D_x}, \boldsymbol{D_y}, \boldsymbol{D_z}).\boldsymbol{v}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{1} & 0 & 0 & \boldsymbol{D_x} \\ 0 & \mathbf{1} & 0 & \boldsymbol{D_y} \\ 0 & 0 & \mathbf{1} & \boldsymbol{D_z} \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x + \boldsymbol{D_x} \\ y' = y + \boldsymbol{D_y} \\ z' = z + \boldsymbol{D_z} \\ 1 \end{bmatrix}$$

Equation 2.6: translation operation

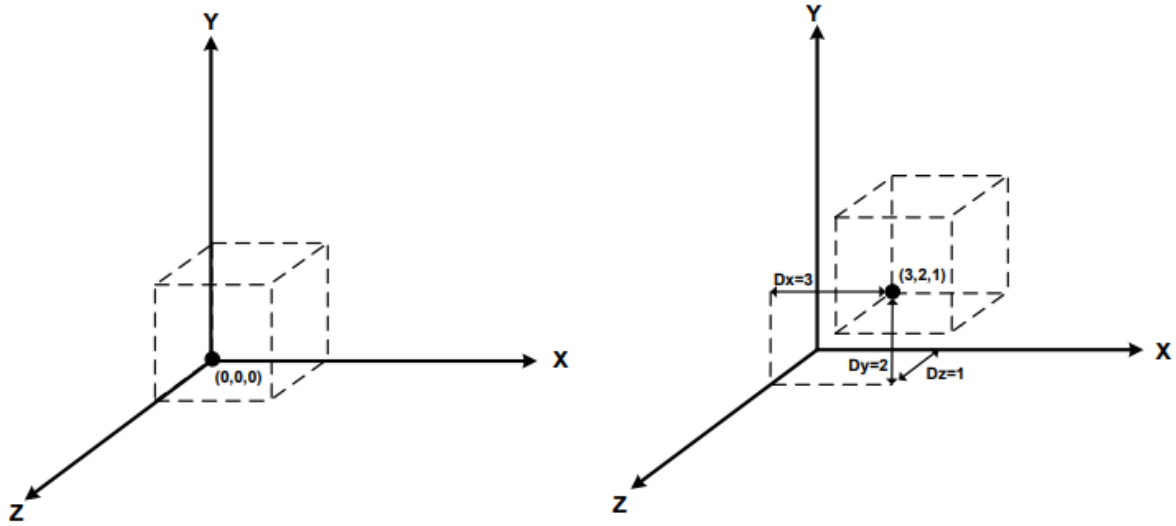

Figure 2.18: Translation of a mesh

[7]

2.2.5 Drawing Lines

Among Different line Drawing algorithm (e.g. DDA, Naïve approach, Bresenham) Bresenham line drawing has been chosen due to its efficiency.

2.2.5.1Bresenham Line Drawing Algorithm

Bresenham's algorithm for scan and converting a line from P1(xl, yl) to P2(x2, y2) with constraints given as x1 < x2 ; 0 <= m < 1 where m is the slope of the line.

```
Initially x = x1 and y = y1.
calculate
dx = x2 - x1, dy = y2 - y1, dT = 2(dy - dx), dS = 2dy;
d = 2dy — dx;
now the line drawing happens here
PutPixel(x, y);
while (x < x2) {
if (d < 0)
d = d + dS;
else
{
d = d + dT;
}
PutPixel(x, y);
}
```

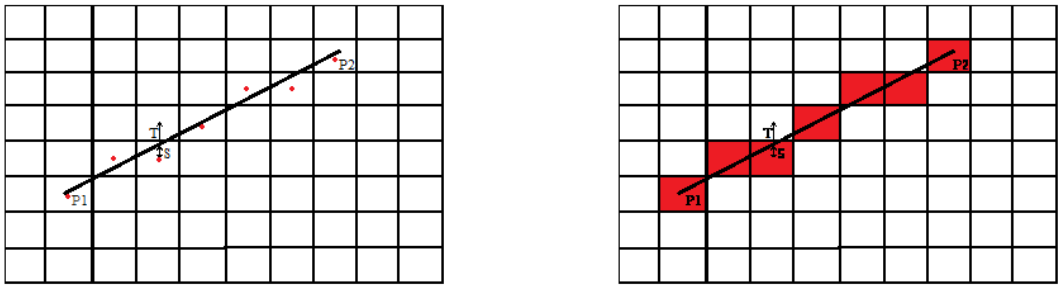

Figure 2.19: Scan conversion of a line using Bresenham's Line Drawing algorithm

2.2.6 Drawing Filled Triangle

A triangle in our preferred colour should be drawn. There are several approaches that may be used to solve this issue, as is frequently the case with computer graphics. We'll create full triangles by seeing them as a collection of horizontal straight line segments that, when joined, form a triangle. Figure 2.16 depicts how one such triangle would appear if each section were visible [15].

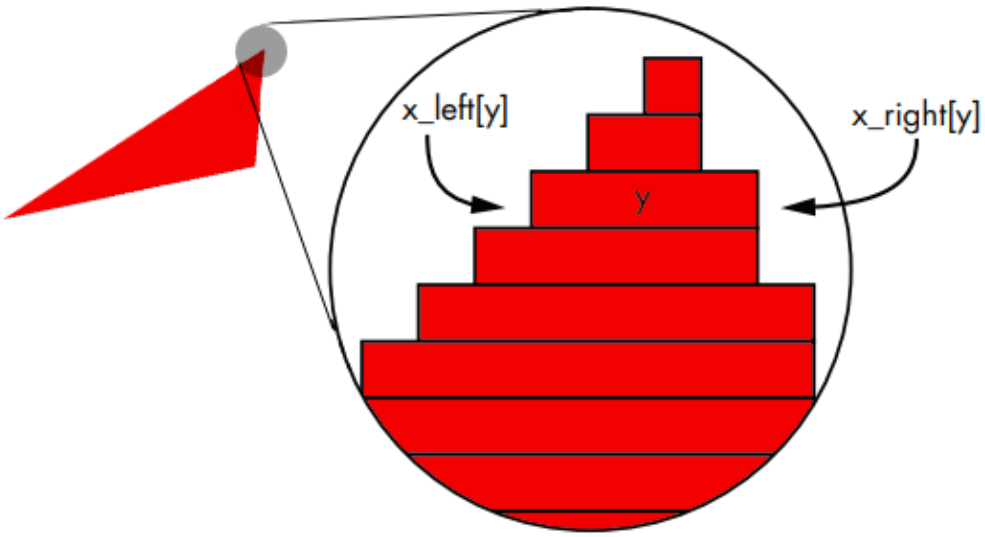

Figure 2.20: Filled triangles by a set of horizontal line segments

A very rough first estimate of what we want to achieve is as follows:

- Iterate every y between the triangle's top and bottom for each horizontal lines
    - Calculate X_Left and X_Right for current y
    - draw_bresenham_line(X_Left, y, X_Right, y)

2.2.7 Perspective Projection

Assume a point at position P ahead of the camera. As seen in Figure 2.17, The spot on the viewport where the camera sees P is designated as P'.

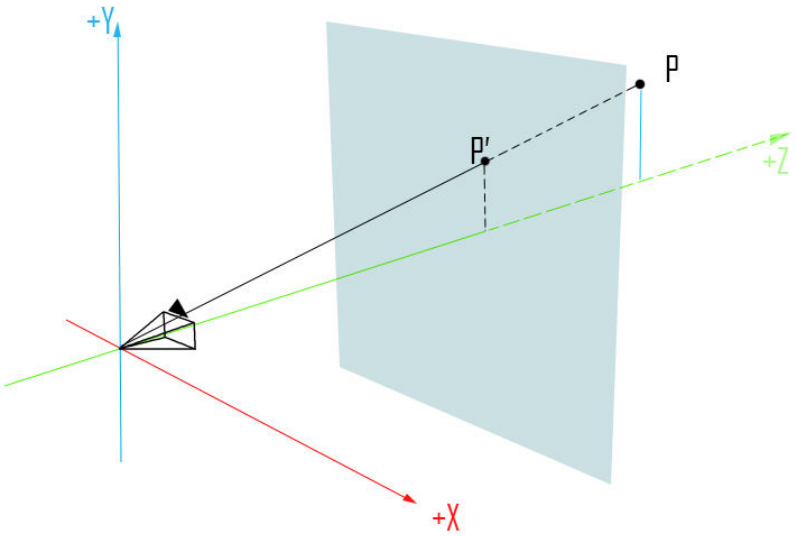

Figure 2.21: Perspective projection, camera sees P via P', which lies on projection plane

To find P', let's physically examine the structure shown in Figure 2.14 from a different angle. Figure 2.18 depicts the layout as seen from the "right," or as if we were standing on the X axis. Z+ indicates a right turn, Y+ indicates an upward movement, and X+ indicates our orientation.

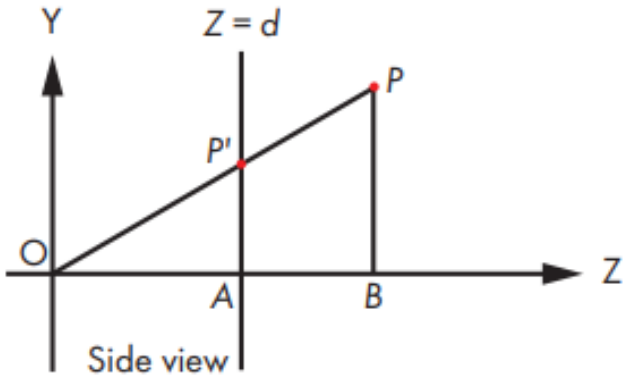

Figure 2.22: the perspective projection setup, right view

[15]

This graphic also depicts the points A and B, which aid our reasoning, in addition to O, P, and P′.

Given that P′ was designated as a point on the viewport and are aware that the viewport is a part of the plane Z = d, we know that P'_z = d.

The triangles P′A and PB, OP and OP ′, and OA and OB may also be demonstrated to be similar due to the parallel alignment of their respective sides. They are equivalent in terms of side dimensions as a result.

$$\frac{P'A}{OA} = \frac{PB}{OB}$$

Equation 2.7: Similarity property of two Triangle

From this we get

$$P'A = \frac{PB\,.OA}{OB}$$

Equation 2.8: Similarity property of two Triangle

Each segment's (signed) length in that equation corresponds to a coordinate for a known or desired point: |P′A| =P'_y, |PB| =P_y, |OA| = P'_z= d, and |OB| =P_z. The result of substituting these in the equation is

$$P'_y = \frac{P_y.d}{P_z}$$

Equation 2.9: Projection no Y axis of the view plane

We may determine that by applying identical triangles once more in the same manner.

$$P'_x = \frac{P_x.d}{P_z}$$

Equation 2.10: Projection on X axis of the view plane

We currently know P ″s three coordinates. This is how a point is projected from 3D space into a 2D space.

Chapter 3

METHODOLOGY

This chapter discusses the design methodology, from the algorithmic frontend to hardware backend. The uses of design tools for creation of software and hardware are treated, and how they are combined to achieve great design. This chapter also includes description of functionalities of different modules of the project. Xilinx Vivado Design Suite has been used for synthesis, implementation and bit-stream generation. From Various FPGA models, Artix7 has been chosen as the target device.

3.1 Proposed Architecture

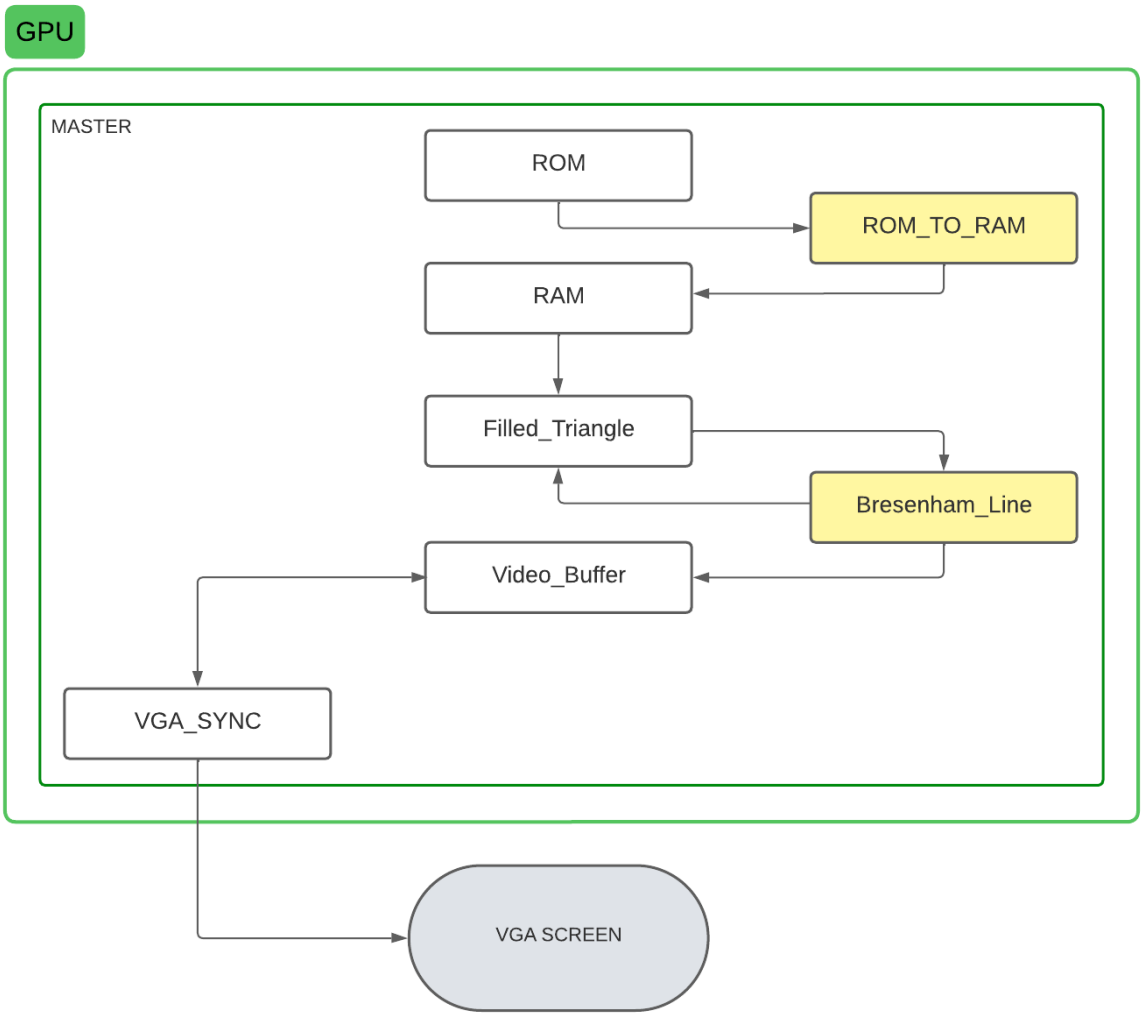

Figure 3.1: The proposed architecture

The Proposed Architecture contains 8 modules named as ROM, RAM, ROM_TO_RAM, Filled_Triangle, Bresenham_Line, Video_Buffer, VGA_SYNC, and MASTER. ROM contains vertex buffer data. ROM_TO_RAM module fetch data from ROM and Writes To RAM, Filled_Triangle module gets data from Rom and it calls Bresenham_Line module to draw horizontal line in triangles. And also each points in horizontal lines is written to the Video_Buffer memory which is looked up by VGA_SYNC circuit that draws coloured pixel in VGA screen. The communication between modules is controlled by MASTER module.

3.2 Implementation of Graphics Processing Modules

The hardware implementation of the mentioned modules in the Proposed Architecture section is described here in terms of their functionality.

3.2.1 ROM Module

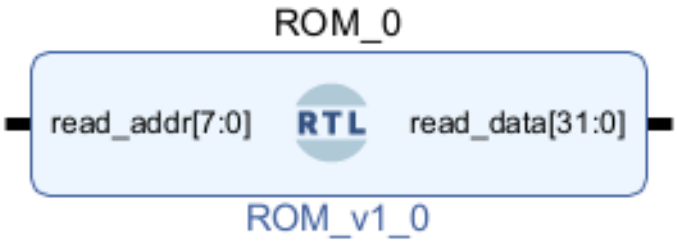

Figure 3.2: ROM module block diagram

The ROM module stores the vertex buffer data. The vertex data contains information about a 3D model. It contains the vertex positions (Vx, Vy, Vz) of each triangle which can be accessed by the wire read_addr which is a 8 bit long bus. The data stored are 32 bit long. It can only store data, and can only be read from. Any write operation in this module is invalid.

3.2.2 RAM Module

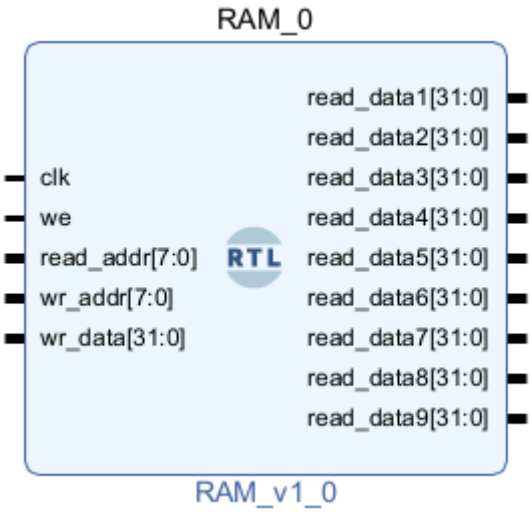

Figure 3.3: RAM module block diagram

The RAM module can do both read and write operations. It is feed with data congaing one triangle at a time from ROM by ROM_To_RAM module. Once Rasterization of one triangle is done its feed with the next triangle from ROM.

3.2.3 ROM_TO_RAM Module

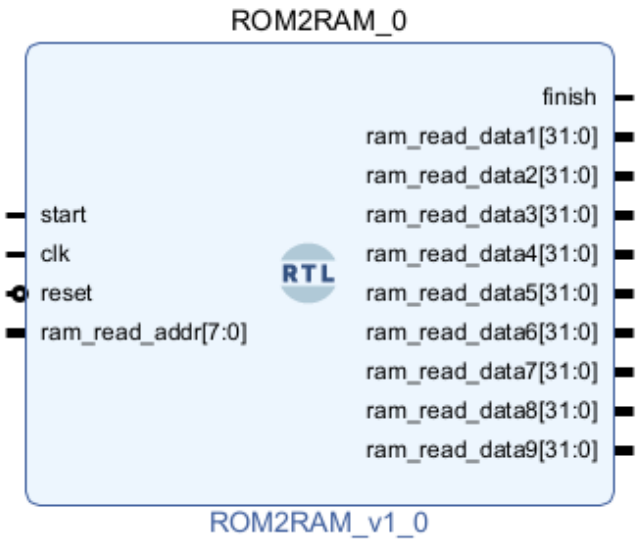

Figure 3.4: ROM_TO_RAM module block diagram

The ROM_TO_RAM module transfers data from ROM to RAM one smallest model data unit (vertex component Vx, Vy or Vz of the triangles of the model) at a time. The working procedure is described by the FSM diagram in Figure 3.5.

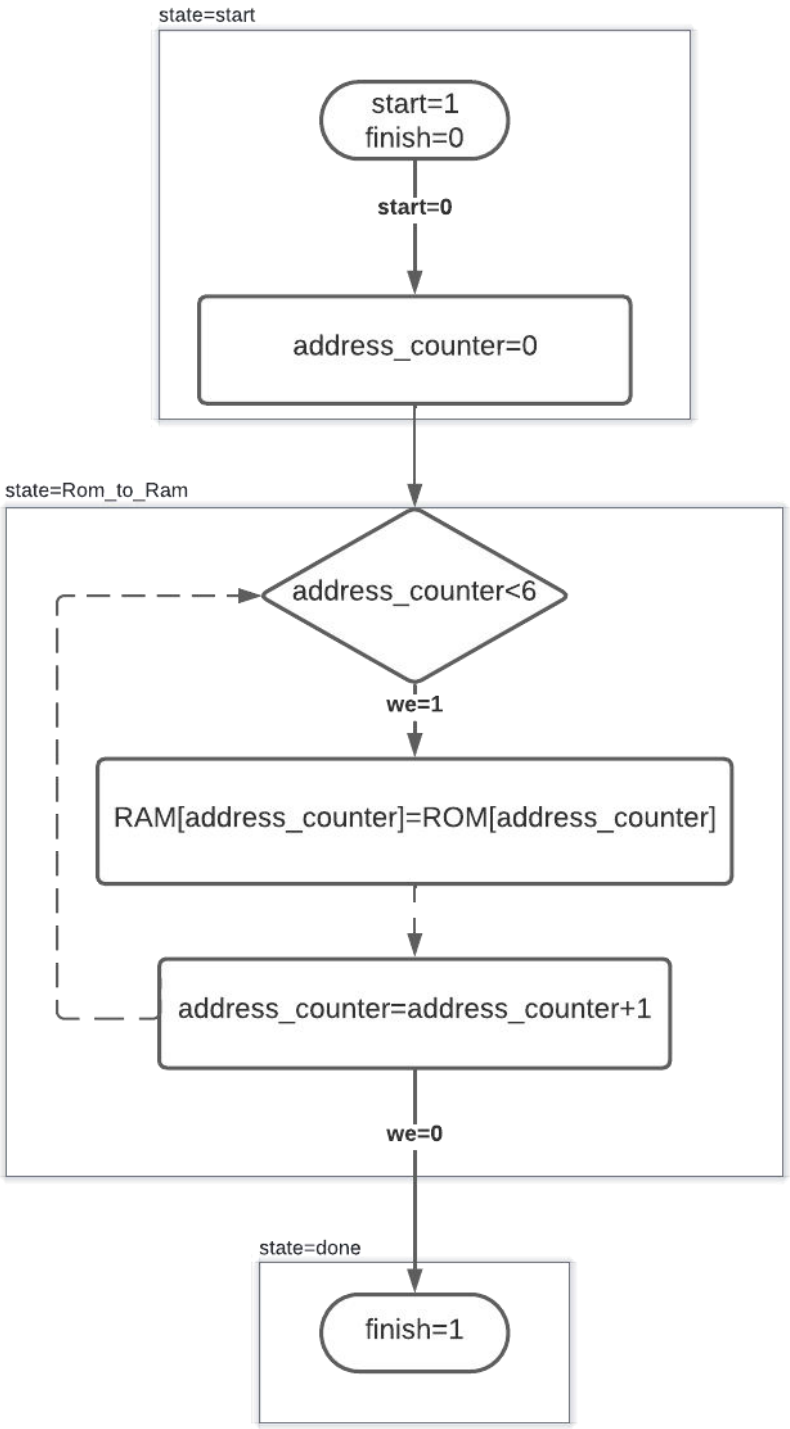

Figure 3.5: FSM diagram of ROM_TO_RAM module

3.2.4 Bresenham_Line Module

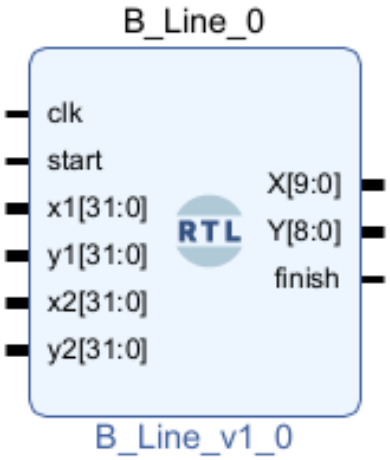

Figure 3.6: Bresenham_Line Module block diagram

This module takes two end points of a line and outputs interpolated points between the two ends of that line using bresenham line drawing algorithm. The working procedure of Bresenham_Line module is demonstrated by the FSM diagram in Figure 3.7.

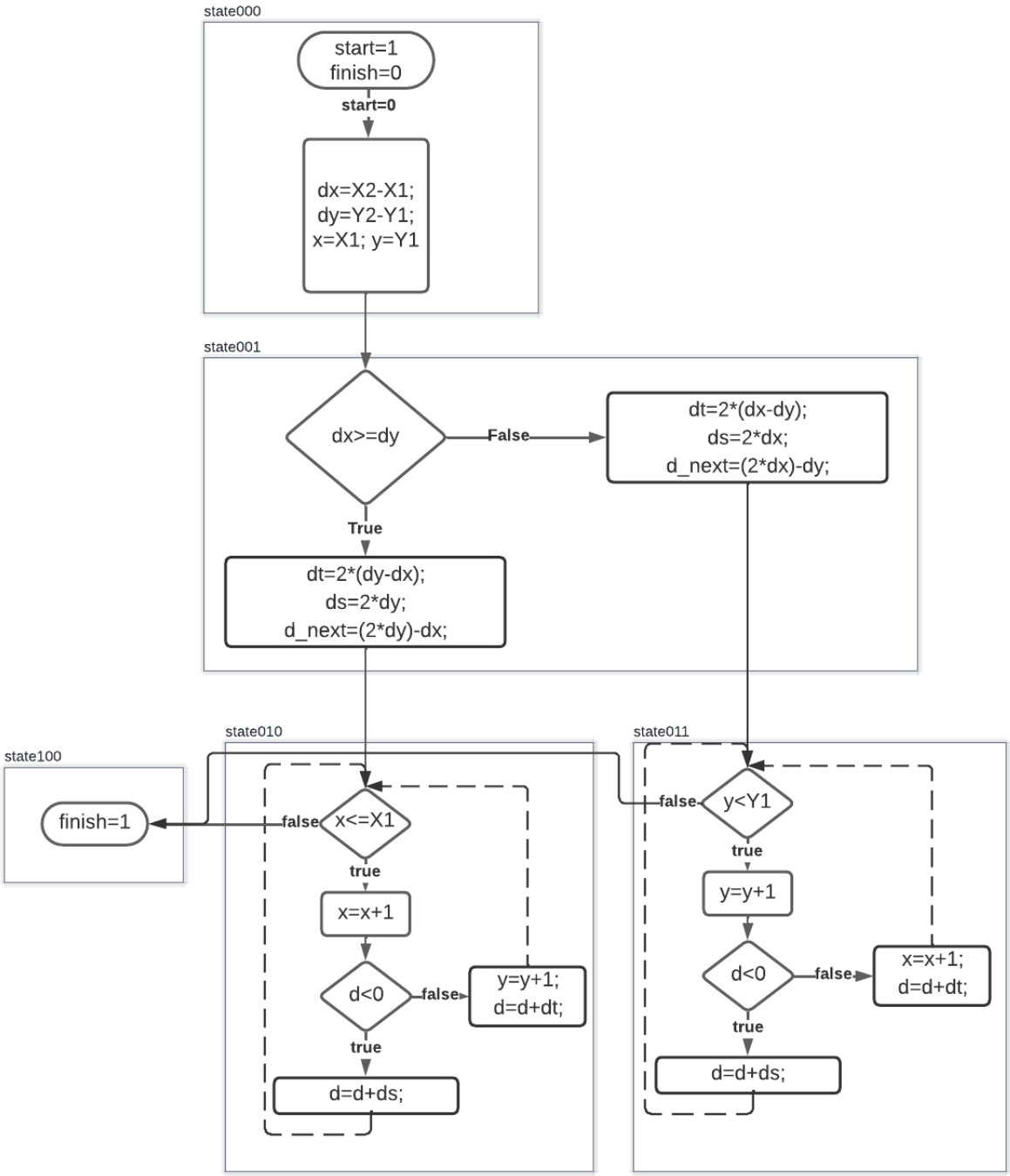

Figure 3.7: FSM diagram of Bresenham_Line module

3.2.5 Filled_Triangle Module

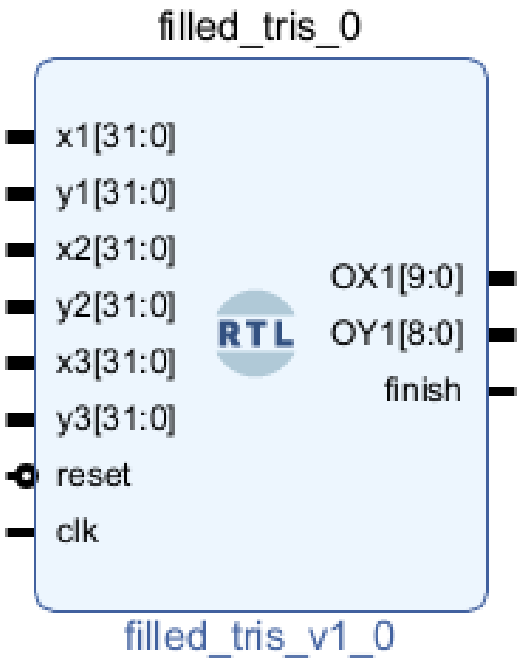

Figure 3.8: Filled_Triangle module block diagram

The filled triangle module is feed with the three vertices of each triangle of the model. It outputs the filled pixel coordinates inside the specified triangle, which is written into video buffer memory for displaying in the screen later on. The working procedure of Filled_Triangles module is demonstrated in the FSM diagram in Figure 3.9

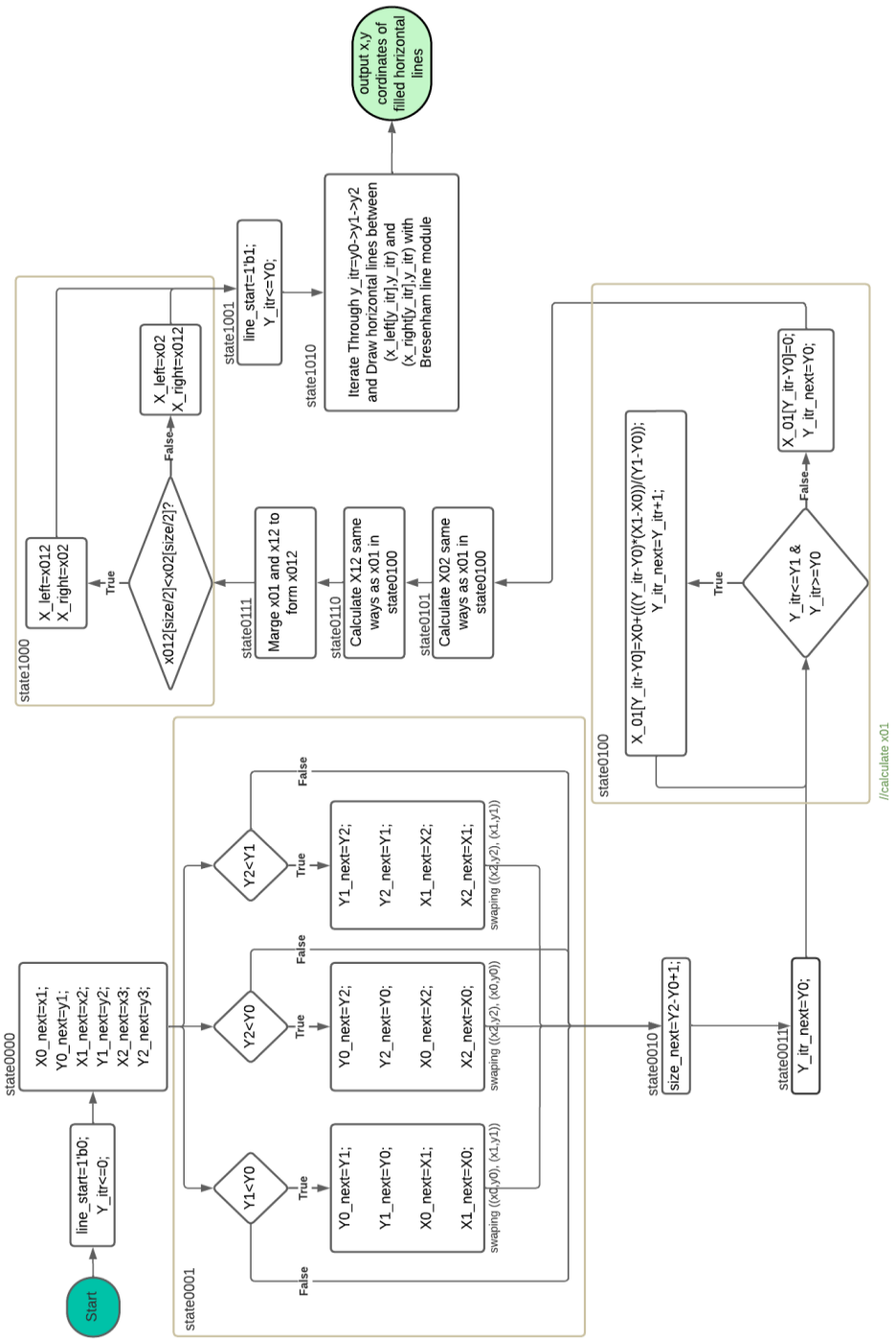

Figure 3.9: Filled_Triangle FSM digram

3.2.6 Video_Buffer Module

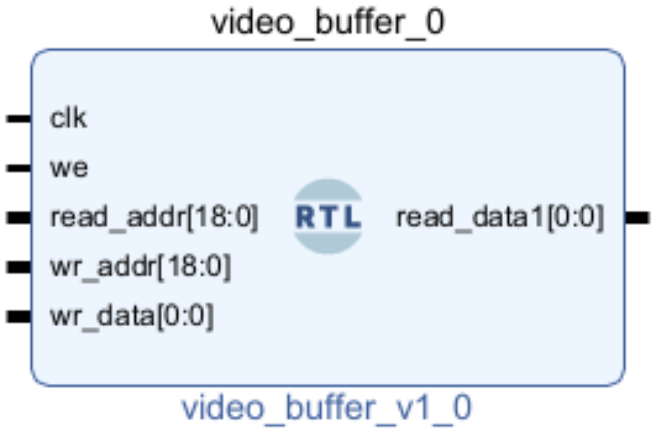

Figure 3.10: Video_Buffer module block diagram

The video Buffer module is feed with the pixel that needs to be drawn on the display. Each pixel position is encoded with a unique address scheme as

Pixel address = {P_X[9:0], P_Y[8:0]}

Where P_X and P_Y are the pixel's X and Y coordinates consecutively.

Now if a pixel needs to be coloured the corresponding address at the memory for the pixel is set to 1 as status bit, referring it needs to be coloured. The VGA_Sync circuit checks the status of each pixel in video memory. And if the status bit is 1, it is coloured in the screen.

3.2.7 VGA_SYNC Module

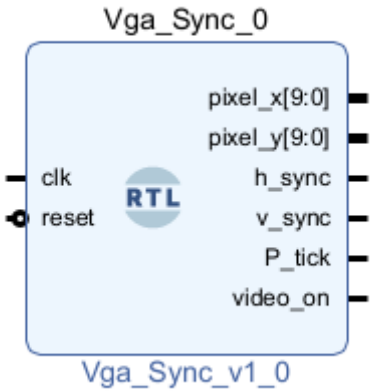

Figure 3.11: VGA_SYNC module block diagram

The VGA_Sync Module interrupts the VGA monitor with h_sync [Horizontal Sync] signal and v_sync [Vertical Sync] signal. And it iterates thorugh every pixel in the screen with it pixel_x and pixel_y iterator and checks video buffer memory for each pixels status bit. If the status bit is 1 then the RGB value will be a pre-specified colour code which is handled by the master module discussed later.

3.2.8 MASTER Module

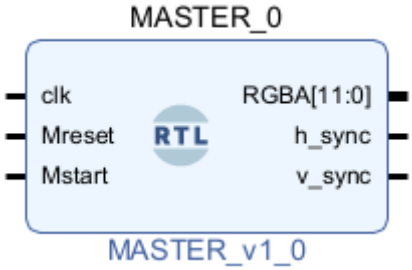

Figure 3.12: MASTER module block diagram

The master module has been designed to utilize all the sub modules discussed before. It organizes each module and outputs final model in the display screen

The work flow of MASTER module is depicted in figure 3.13 by a FSM diagram

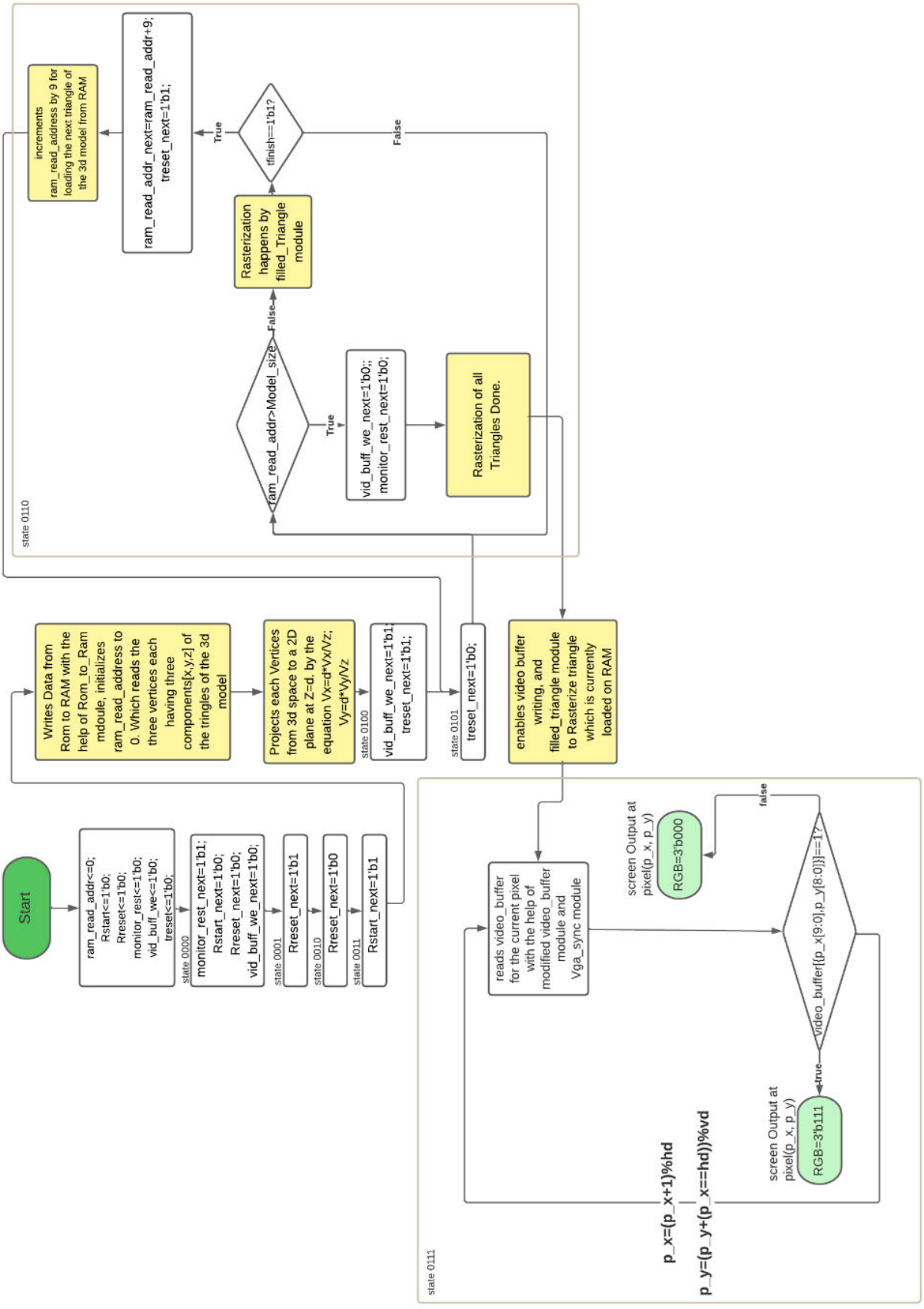

Figure 3.13: MASTER module FSM diagram

CHAPTER 4

RESULTS ANALYSIS

In this chapter Test-Bench (GTKWave simulation) and Terminal output results of each of the modules is shown and also at the end, final output of the Graphics Processor is shown.

4.1 Environment Setup

For coding Visual Studio Code has been used. For simulation GTKWave is used. And logic synthesis, implementation, bit stream generation has been done in Vivado Design Suite ML Edition software. Targeted device was Nexys 4 board which is equipped with Artix 7 FPGA. The PC where the software were executed, had Windows 10 as the operating system.

4.2 ROM Module Results

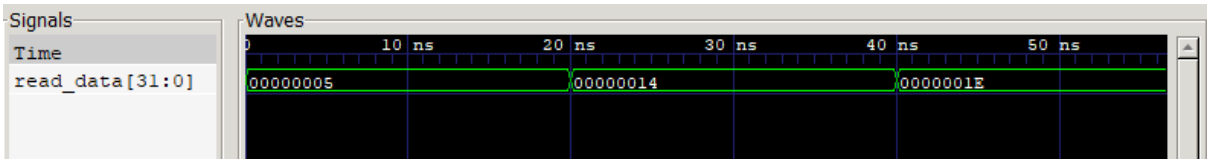

Figure 4.1: ROM module simulation results

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

PS C:\Users\USER\Desktop\vivado\linefiles> make
iverilog -o out.vpp ROM.v #ROM.v RAM.v ROM_TO_RAM_LOADER.v LINE_MODULE.v Vga_Sync.v master_circ.v video_buffer.v
vvp out.vpp
addr= 00000000
data=          5

addr= 00000001
data=         20

addr= 00000010
data=         30

#gtkwave test.vcd
PS C:\Users\USER\Desktop\vivado\linefiles>
```

Figure 4.2 ROM module terminal output

As discussed previously, the ROM module can only be read from, the operations performed on it is only read operation. It outputs data stored in those provided read address.

4.3 RAM Module Results

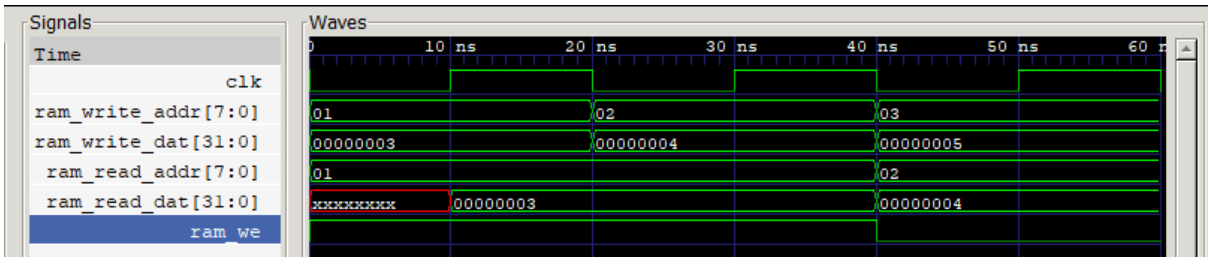

Figure 4.3: RAM module simulation results

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

PS C:\Users\USER\Desktop\vivado\linefiles> make
iverilog -o out.vpp RAM.v #ROM.v RAM.v ROM_TO_RAM_LOADER.v LINE_MODULE.v Vga_Sync.v master_circ.v video_buffer.v
vvp out.vpp
VCD info: dumpfile test.vcd opened for output.
wr_addr= 00000001
wr_data=          3
read_addr= 00000001
read_data=          x
wr_addr= 00000001
wr_data=          3
read_addr= 00000001
read_data=          3
wr_addr= 00000010
wr_data=          4
read_addr= 00000001
read_data=          3
wr_addr= 00000011
wr_data=          5
read_addr= 00000010
read_data=          4
RAM.v:68: $finish called at 60000 (1ps)
gtkwave test.vcd
```

Figure 4.4: RAM module terminal output

The RAM module can be read from and also written to. The results shows it was successful to write data 3, 4, 5 in the memory location of 1, 2, and 3 consecutively and was read from the addresses at 1 and 2.

4.4 ROM_TO_RAM Module Results

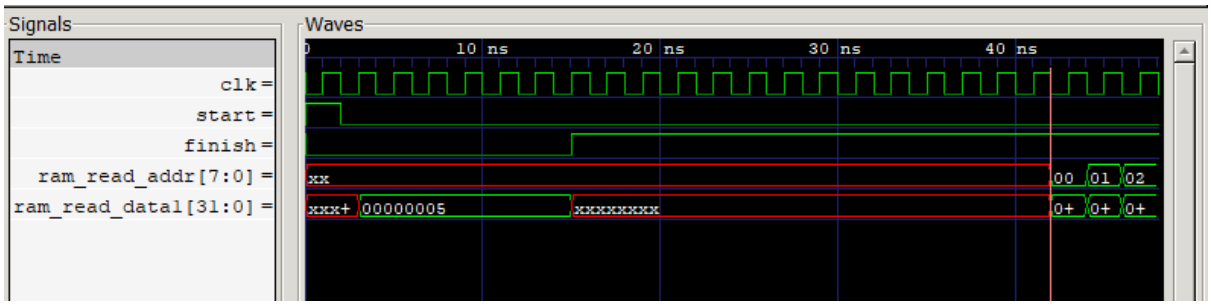

Figure 4.5: ROM_TO_RAM module simulation results

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

iverilog -o out.vpp ROM_TO_RAM_LOADER.v ROM.v RAM.v #ROM_TO_RAM_LOADER.v LINE_MODULE.v Vga_Sync.v master_circ.v video_buffer.v
vvp out.vpp
load_start=1 | load_finish=0
ram_read_addr=  x
ram_data=         x

load_start=0 | load_finish=0
ram_read_addr=  x
ram_data=         x

load_start=0 | load_finish=0
ram_read_addr=  x
ram_data=         5

load_start=0 | load_finish=1
ram_read_addr=  x
ram_data=         x

load_start=0 | load_finish=1
ram_read_addr=  0
ram_data=         5

load_start=0 | load_finish=1
ram_read_addr=  1
ram_data=        20

load_start=0 | load_finish=1
ram_read_addr=  2
ram_data=        30

ROM_TO_RAM_LOADER.v:136: $finish called at 48000 (1ps)
#gtkwave test.vcd
PS C:\Users\USER\Desktop\vivado\linefiles>
```

Figure 4.6: ROM_TO_RAM module terminal output

From the Results it is clear that the RAM_TO_RAM module was successfully able to read from ROM and write to RAM. After starting it read and wrote data one by one from ROM to RAM. And after it was done it was instructed to show what it has stored in specific address in the RAM provided by ram_read_addr signal, which it showed by the signal ram_read_data1.

4.5 Bresenham_Line Module Results

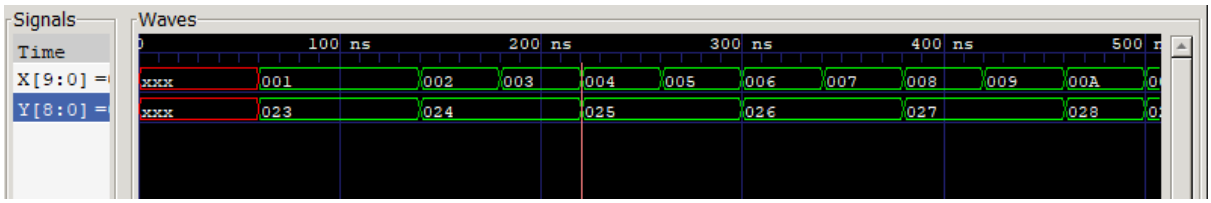

Figure 4.7: Bresenham_Line module simulation results

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

PS C:\Users\USER\Desktop\vivado\linefiles> make
iverilog -o out.vpp LINE_MODULE.v #ROM.v RAM.v ROM_TO_RAM_LOADER.v LINE_MODULE.v Vga_Sync.v master_circ.v video_buffer.v
vvp out.vpp
VCD info: dumpfile test.vcd opened for output.
x=   x, y=  x |finish=0
x=   1, y= 35 |finish=0
x=   2, y= 36 |finish=0
x=   3, y= 36 |finish=0
x=   4, y= 37 |finish=0
x=   5, y= 37 |finish=0
x=   6, y= 38 |finish=0
x=   7, y= 38 |finish=0
x=   8, y= 39 |finish=0
x=   9, y= 39 |finish=0
x=  10, y= 40 |finish=0
x=  11, y= 41 |finish=0
x=  11, y= 41 |finish=1
LINE_MODULE.v:156: $finish called at 4040000 (1ps)
gtkwave test.vcd
```

Figure 4.8: Bresenham_Line module terminal output

The results of this module shows that given two end points (1, 35), (11, 41) of a line it was able to interpolate intermediate points. (1, 35), (2, 36), (3, 36), (4, 37) ... (11, 41).

4.6 Filled_Triangle Module Results

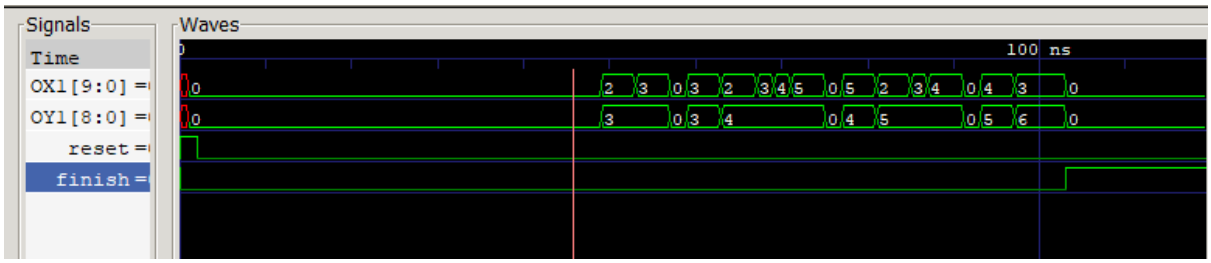

Figure 4.9: Filled_Triangle module simulation results

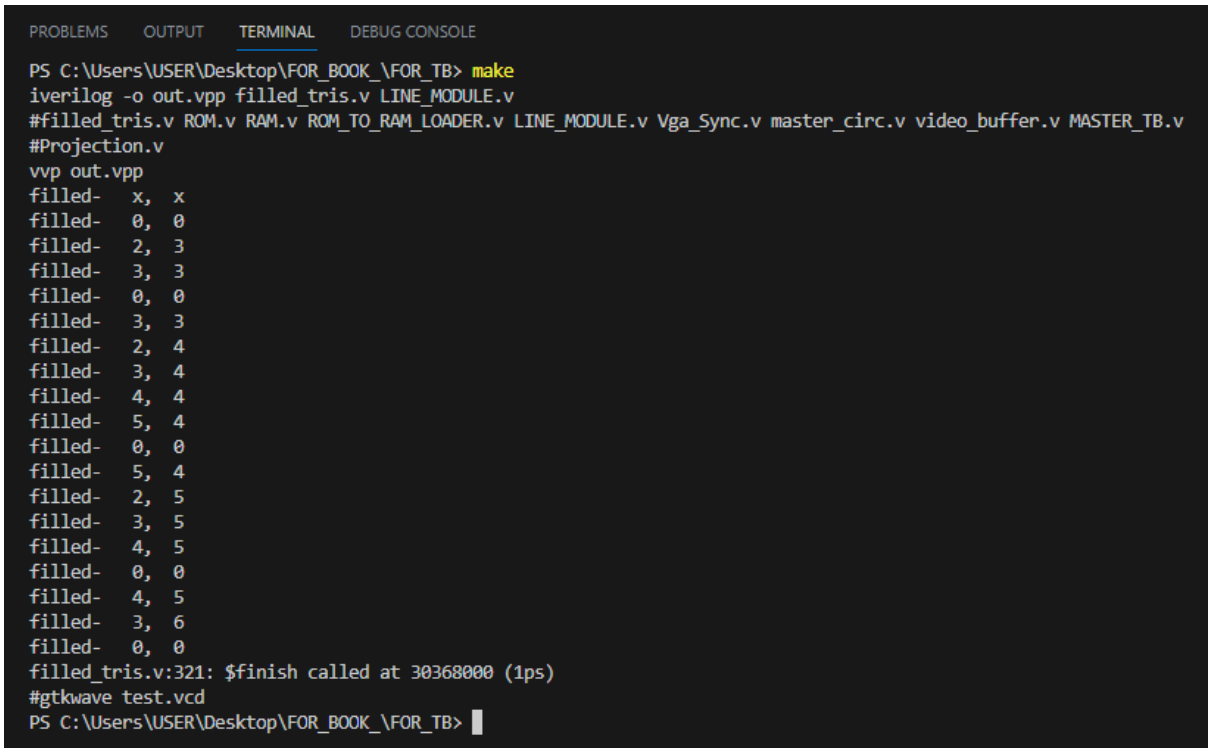

Figure 4.10:Filled_Triangle module terminal output

As we can see from the outputs and simulation result of Filled_Triangle module it was able to fill a triangle with pixel inside the triangle. It was provided with three vertices of a triangle (2, 2), (5, 4), (3, 6). It provided the filled pixels inside it. It also filled (0, 0) pixel but it was a default case.

4.7 Final Graphics Processor Output

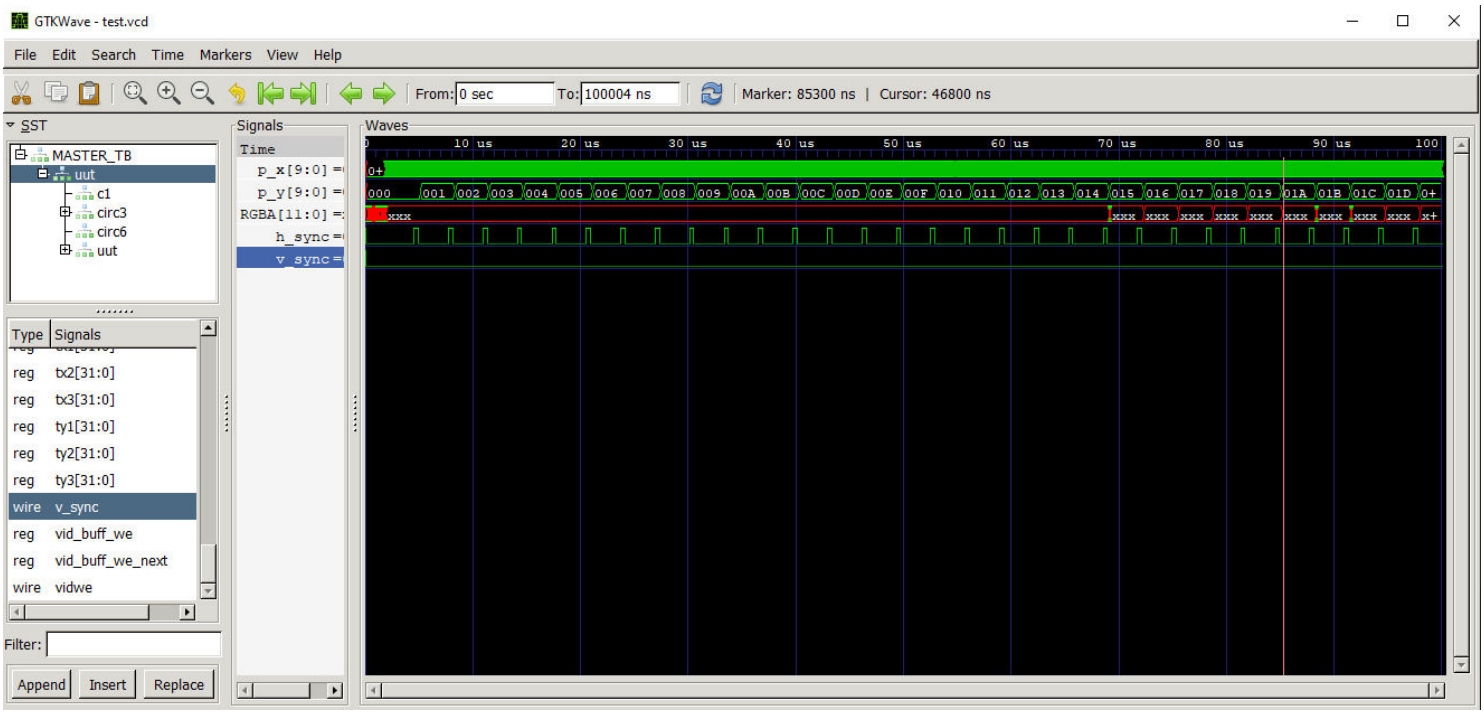

Figure 4.11: Simulation result of the Graphics Processor in GTKWave

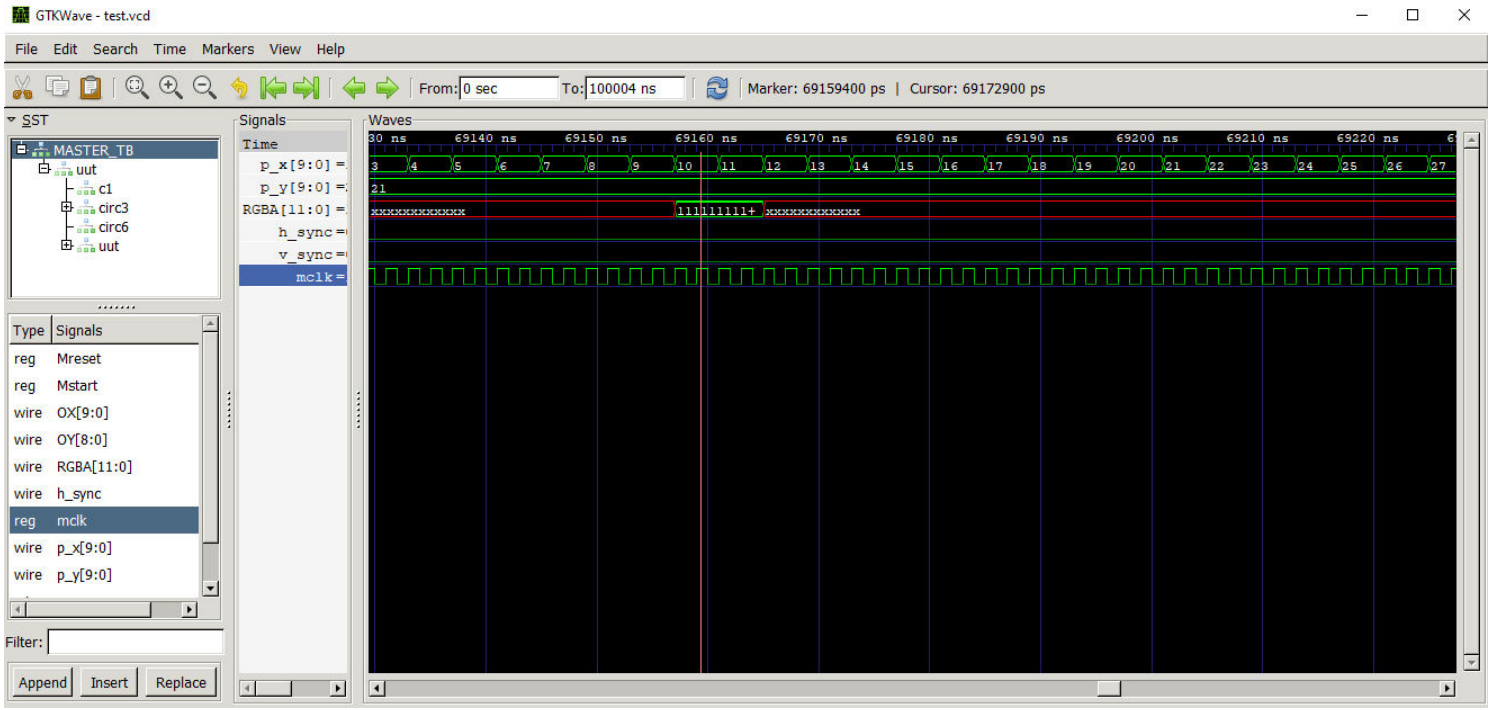

Figure 4.12: Simulation result of the Graphics Processor in GTKWave (zoomed in)

As we ca see the GPU can generate RGB values at different pixels according to the rendered model. It also interfaces the VGA monitor using the h_sync and v_sync signal.

In the above simulation figure, it is observable that the RGB was 111111111111 at pixel (10,21) and (11,21). And h_sync was set to 1 at end of the display region in the in x axis. And v sync was set to 1 at end of the (y axis & x axis) it is not observable as the simulation was not run that long

CHAPTER 5

CONCLUSION & FUTURE WORK

5.1 Conclusion

The dynamic integration of FPGA technology into the realm of 3D graphics rendering holds immense promise for both real-time performance enhancements and educational exploration. In conclusion a graphics processing hardware unit was designed in FPGA for educational purpose. In this thesis paper study about computer graphics was done first, then about FPGA device and its properties. Finally graphics rendering algorithm was implemented on FPGA. It was shown that the implementation was able to render 3D models. All simulation results are verified.

The adaptability of FPGAs, coupled with their parallel processing capabilities, offers a unique platform for implementing efficient rendering pipelines. Furthermore, leveraging FPGA-based rendering as an educational tool provides learners with hands-on experience in merging computer graphics theories with hardware design realities. As advancements in FPGA technology continue to unfold, this synergy between 3D graphics and FPGA implementation is poised to catalyse innovation, offering new dimensions in learning and real-time rendering performance alike.

5.2 Future Work

The future scopes includes floating point arithmetic unit implementation for higher precision and simulating effects of different lighting (ambient, diffuse, specular) on 3d models.

References

[1] A. Clements, "Computer Architecture education," *IEEE Micro,* vol. 20, no. 3, pp. 10-12, 2000.

[2] T. Angelia, "Rendering Techniques in Computer Graphics," 20 7 2020. [Online]. Available: https://medium.com/@tifangel/rendering-techniques-in-computer-graphics-504e6134fea4.

[3] "An Overview of the Ray-Tracing Rendering Technique," 2023. [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview.html. [Accessed 09 08 2023].

[4] "Rasterization: a Practical Implementation," [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html. [Accessed 2023 08 2023].

[5] V. Kasik, A. Kurecka and P. Pospech, "3D graphics processing unit with VGA output," *IFAC Proceedings Volumes,* vol. 46, no. 28, pp. 388-393, 2013.

[6] J. Schmittler, S. Woop, D. Wagner, W. J. Paul and P. Slusallek, "Realtime ray tracing of dynamic scenes on an FPGA chip," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware,* 2004.

[7] J. R. Warner, *Real Time 3-D Graphics Processing Hardware Design using Field-Programmable Gate Arrays,* Pittsburgh, 2009.

[8] "What is an FPGA? field programmable gate array," AMD, [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed 09 08 2023].

[9] "Nexys A7 - Digilent Reference," [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/start. [Accessed 09 08 2023].

[10] National Instruments, "FPGA fundamentals: Basics of field-programmable gate arrays," [Online]. Available: https://www.ni.com/en-us/shop/electronic-test-instrumentation/add-ons-for-electronic-test-and-instrumentation/what-is-labview-fpga-module/fpga-fundamentals.html. [Accessed 09 08 2023].

[11] "What is FPGA? FPGA Basics, Applications and Uses: Arrow.com," 13 10 2022. [Online]. Available: https://www.arrow.com/en/research-and-events/articles/fpga-basics-architecture-applications-and-uses#:~:text=A%20basic%20FPGA%20architecture%20(Figure,the%20FPGA%20and%20external%20devices.. [Accessed 09 08 2023].

[12] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array#/media/File:FPGA_cell_example.png.

[13] P. P. Chu, FPGA prototyping by VHDL examples: Xilinx spartan-3 version, Hoboken, NJ: Wiley-Interscience, 2008.

[14] D. K. Tala, "How to write FSM in Verilog?," 01 February 1970. [Online]. Available: https://www.asic-world.com/tidbits/verilog_fsm.html#:~:text=Basically%20a%20FSM%20consists%20of,shown%20in%20the%20figure%20below..

[15] G. Gambetta, Computer graphics from scratch: A programmer's introductrion O 3D rendering, San Francisco: No Starch Press, 2021, pp. 92-96.