Synthesizable VHDL Model of 3D Graphics Transformations

Daniel Mc Keon

BA (Mod.) Computer Science

Final Year Project 2005

Supervisor: Michael Manzke

Acknowledgements

I would like to thank my supervisor, Michael Manzke, for giving me the chance to undertake a hardware engineering project and for his advice and encouragement throughout its duration.
I would also like to thank Hitesh Tewari for his interest at my project demonstration and for being my second reader.
A final thanks to everyone who helped me solve a problem or provided support, you know who you are.

Abstract

Since their emergence only 15 years ago, Field Programmable Gate Arrays, have gone from strength to strength in the domain of reprogrammable hardware. As they improve, they grow in speed, size and ability. All these factors mean FPGAs are becoming more useful in areas where fixed hardware was the basis before. One such area is in 3D Graphic rendering, where research is underway as to how programmability and specifically reprogrammable FPGAs can help to improve the performance of graphic processing units.

One area FPGAs can assist is through the handling of complex algorithms on inexpensive, dynamically reconfigurable hardware. In my project I attempt to put this idea into practice by implementing a design for the transformations and projections of 3D vectors onto a Spartan 3 FPGA. I attempt to do this through a hardware description language (VHDL) and explore the problems, solutions and alternatives faced when approaching such a project.

Contents

1 Introduction **1**

2 Background **2**

2.1 Personal Background 2
2.2 Tools 3
2.2.1 Software 3
2.2.2 Hardware 5
2.3 3D Graphics 7
2.4 3D Graphics on Computers 10
2.4.1 3D Graphic Pipeline 10
2.4.2 3D Geometry Pipeline 10
2.5 3D Graphic Transformations 12
2.5.1 Transformation Matrices 12
2.5.2 Matrix Multiplication 13
2.5.3 Scaling Matrix 13
2.5.4 Rotation Matrices 14
2.5.5 Homogenous Co-ordinates 14
2.6 3D Graphic Projections 16
2.6.1 Orthographic Projection 16
2.6.2 Perspective Projection 17
2.7 3D Graphic Hardware 22
2.7.1 Graphics Processing Unit 22
2.7.2 FPGAs as an Alternative 23
2.7.3 FPGAs as an Assistance 24

2.8 IEEE Single Precision Floating Point Numbers 24

3 Design and Implementation **27**

3.1 Binary Multiplier 27

3.1.1 Design 27

3.1.2 Implementation 27

3.1.3 Conclusion 29

3.2 Behavioural Design 29

3.2.1 Design 29

3.2.2 Implementation 30

3.2.3 Conclusion 31

3.3 Memory Load Design 32

3.3.1 Design 32

3.3.2 Implementation 35

3.3.3 Conclusion 35

3.4 Final Design 35

3.4.1 Design 35

3.4.2 Implementation 36

3.4.3 Conclusion 40

3.5 Compact Design 41

3.5.1 Design 41

3.5.2 Implementation 42

3.5.3 Conclusion 43

4 Evaluation **44**

4.1 Individual Components 44

4.2 Integrated Components 52

5 Future Work and Conclusions **61**

5.1 Completion 61

5.2 Expansion 63

5.3 Experiences 63

5.4 Conclusion 65

5.5 Attached CD 65

List of Figures

2.1 Xilinx Spartan 3 XC 3S1000 and its Block RAM[1] 6
2.2 XSA-3S1000 Board 7
2.3 Axes of 3D space [2] and a Vector in Matrix Form 8
2.4 Wireframe Cows [3] 9
2.5 Shaded and Lighted Cow [3] 9
2.6 3D Geometry Pipeline[4] 10
2.7 Viewing Frustum[5] 11
2.8 Identity Matrix 13
2.9 Matrix Multiplication 13
2.10 3x3 Scale Matrix 14
2.11 3x3 X,Y and Z axis Rotation Matrices 14
2.12 4x4 Translation Matrix and Vector matrix with Homogeneous W . 15
2.13 4x4 Scale Matrix with Homogeneous W 15
2.14 4x4 Rotation Matrices with Homogeneous W 16
2.15 Orthographic Projection 17
2.16 Vanishing Points in Perspective Projection [6] 19
2.17 Perspective Projection Matrix 20
2.18 Perspective Projection[7] 21
2.19 Multiplying the Perspective Projection Matrix by an input Vector . 21
2.20 Dividing across by Homogenous W 22
2.21 IEEE Single Precision Floating Point Numbers 25
2.22 IEEE Single Precision Floating Point Example 26

3.1 Mano and Kime's Binary Multiplier Block Diagram 28
3.2 Components of My Binary Multiplier 29

3.3 Behavioural Design 30
3.4 Xilinx 3x3 Matrix Multiplier Diagram[8] 33
3.5 Two Stage Addition 33
3.6 Multiplication and Addition Stages 34
3.7 4x4 Transform Matrix for Translation, Scaling and Rotation around the Z Axis 37
3.8 Final Perspective Vector 40

4.1 Multaddtest Test Bench Waveform - Part 1 46
4.2 Multaddtest Test Bench Waveform - Part 2 47
4.3 Testtoadd Test Bench Waveform - Part 1 49
4.4 Testtoadd Test Bench Waveform - Part 2 50
4.5 Final Test Bench Waveform - RVST = 0011 - Part 1 53
4.6 Final Test Bench Waveform - RVST = 0011 - Part 2 54
4.7 Final Test Bench Waveform - RVST = 1100 - Part 1 56
4.8 Final Test Bench Waveform - RVST = 1100 - Part 2 57
4.9 Final Test Bench Waveform - RVST = 1111 - Part 1 59
4.10 Final Test Bench Waveform - RVST = 1111 - Part 2 60

List of Tables

2.1 IEEE Single Precision Floating Point Representations 26

Chapter 1

Introduction

The aim of my project is to create a design to implement vector transformations and projections through the VHDL hardware description language. This design should then synthesize and successfully load onto a reconfigurable Field Programmable Gate Array for testing and evaluation purposes and hopefully aid in the current research being undertaken by Trinity College in cluster based graphics processing.

In this paper I document:

- the research undertaken to approach this project
- the reasoning behind my project
- the many approaches I took to reach my goal
- the problems faced along the way and how I solved them
- the learning process and experiences I drew from my project

From my project I hoped to evaluate the ability of an FPGA to perform complex graphical computations and collaborate with the CPU and GPU with an aim to improve the performance of the system as a whole. I also hoped to gain experience in hardware engineering and individual projects in general.

Chapter 2

Background

In this section I will explain why I undertook this project, the software and hardware used in its duration and the many topics I researched before approaching it practically. Although I tried to find dedicated books relevant to my project, this was difficult as it spanned many subjects such as Linear Algebra, Geometry and 3D graphics. Instead I turned to the internet which proved to be an indispensable, but sometimes frustrating source. Although the amount of interesting information and level of detail I have encountered in the duration of my project could fill books, I will try to keep this section as straight forward and relevant to my project as possible.

2.1 Personal Background

Before I chose this project, I had to decide which area of computer science I would specialize in. I reflected on past subjects and the projects and labs that went with each one and decided that hardware subjects and projects, with their practical, hands on approach, appealed to me more than the software programming side of computer science. I also researched careers and discovered that hardware design was a very interesting occupation and one which I felt I would like to pursue.

It is for this reason I decided to choose Integrated Systems Design as a subject and a final year project based around hardware engineering. I felt that with my comfortable knowledge of the VHDL language and good exam results in com-

puter architecture subjects, as well as the information I would learn from labs and lectures in Integrated Systems Design, a chip design project would be an ideal way to experience, first hand, what it would be like to work as a hardware design engineer.

The problem I then faced was that none of the hardware projects offered for this year immediately appealed to me. I did not know then how interesting this project would turn out to be. After talking to many lecturers about other ideas, but coming up with nothing, Michael Manzke finally encouraged me to take on his ”Synthesizable VHDL Model of 3D Graphic Transformations” project. My only previous encounter with 3D Graphics was with Open GL during my second year C++ Programming classes, leaving me with a lot of research on my hands. I had long since decided that C++ programming was not for me but after only a few days researching what went on behind the Open GL side of things, I realized that 3D Graphics were much more interesting than first expected. I saw and understood what was going on in the background as the 3D images appeared and changed on the screen in front of me. I also discovered the maths behind them were much more approachable than I had anticipated.

Armed with this knowledge, I was excited to begin my project and see what I could do.

2.2 Tools

2.2.1 Software

VHDL

As I have said, the language used for this project was VHDL. This stands for VHSIC Hardware Description Language, which in turn stands for Very-High-Speed Integrated Circuits. All this basically means is that it is a language used to describe the gate logic on a hardware device, originally application specific integrated circuits (ASICs) but later adapted to other hardware such as FPGAs. The language was originally developed by the US department of defence but has evolved into a very popular HDL that complies with IEEE (Institute of Electrical and Electronics

Engineers) standards. Another such language is Verilog HDL.

Both of these languages describe how the hardware is programmed and will act based on input values. This can be done behaviourally, which is when the actual behaviour of a circuit is described at a high level of abstraction. This model can then be simulated with the help of a logic simulator to see how the design would work in hardware. Register transfer level code describes the design of the digital circuits in detail and can be synthesized with a logic synthesis tool, aided with actual chip specifications, and simulated to emulate the exact working of a specific chip. This means that designs can be made and tested in software under nearly the exact conditions as if they were put onto the hardware. This saves time, energy and the expense of putting the design on chip for testing each time. These synthesized designs are also board ready and can be loaded onto hardware such as ASICs or FPGAs (explained below) for testing or marketing. Another way of describing the logic is at gate level, which describes the exact gates which the transistors will form but this is very detailed and usually only used for verification.

Another important aspect of HDL coding I should mention here is instantiation. This is when one component is incorporated into another component, for example if a design is made up of many smaller components which are brought together in one top file to be wired together. The two main methods of doing this are through port mapping or schematics. Schematics allow us to code our components and then draw each one to the screen and use a wiring tool to connect outputs to inputs and so forth until the design layout is complete. The software can then use this schematic to determine how each component is connected. Port mapping is a more HDL oriented way of doing things, in that the connections between components are declared as signals and used to join outputs of one component to inputs of another all through HDL code.

The actual software I used for my design was the Xilinx ISE Webpack, incorporating Project Navigator for laying out and coding projects and Xilinx Synthesis Technology (XST) for synthesising to a vendor specific standard as well as many other tools and functions. I also used Mentor Graphics' ModelSim XE II/Starter 5.8c for simulation and testing of my design. Both of these are limited versions of commercial products and are available freely for student and home use.

Octave

Octave is an excellent piece of open source software for testing a vast range of numerical problems on a command line interface, much like the popular program Matlab. Octave has its own simple to use high-level language or can handle modules written in a wide range of languages like C++ or FORTRAN and because it is open source it is ever expanding to add further compatibility and functionality. I used this program to calculate matrix multiplications with floating point numbers as these are very difficult to work out without such a tool.

LaTeX

LaTeX is a system for producing documents (such as this one) based on Donald E. Knuth's TeX typesetting language. Although looking like a program language at first glance, it is rather easy to comprehend and create professional looking documents with ease.

2.2.2 Hardware

FPGAs

FPGAs are field-programmable gate arrays, which are basically reconfigurable hardware devices that can be loaded with designs with simple to use software. These designs can then be tested and used. If required they can then be replaced with debugged or completely new designs. This is the opposite of a programmable logic device (PLD) or ASIC which are hardcoded by a vendor at manufacture time and therefore cannot be reprogrammed. This gives FPGAs a clear advantage for testing in that, if a mistake is made, money and design time (two of the most important factors in chip design) are not wasted waiting on a manufacturer to produce another batch of chips.

Complex programmable logic devices (CPLDs) are similar to FPGAs but on a much smaller scale and are only capable of producing simple combinational circuits.

FPGAs are made up of configurable logic blocks (CLBs), each with a look-up table, multiplexer and flip flop, to store the logic needed to make up the gates

required of the design. There are also reconfigurable interconnect switches to connect these CLBs together and to the input/output blocks, which surround the chip for interaction with the rest of the board. This is the basic structure of an FPGA though there are many different types and sizes available.

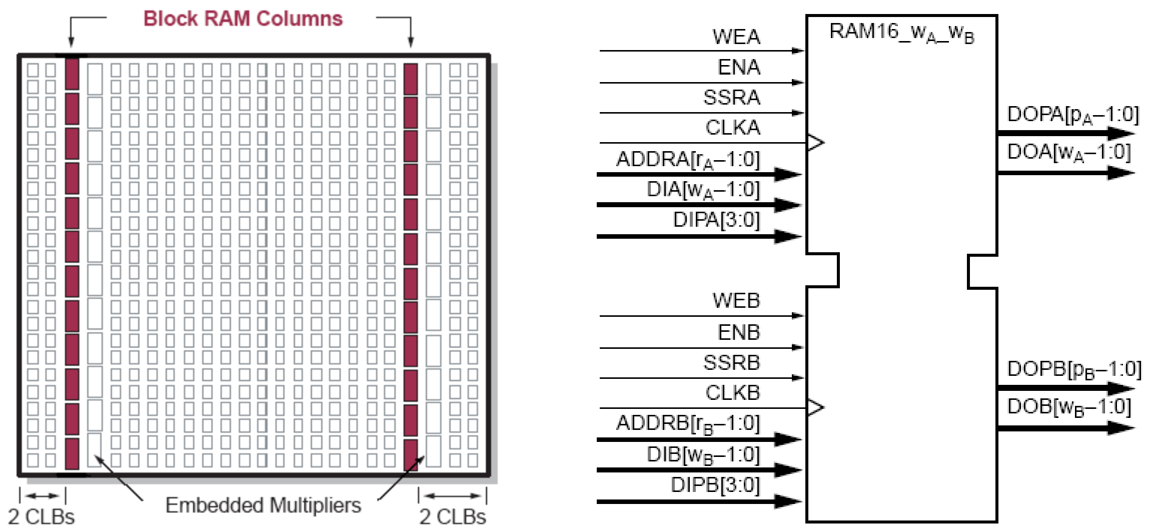


Figure 2.1: Xilinx Spartan 3 XC 3S1000 and its Block RAM[1]

The FPGA that was supplied on our project board was a Spartan 3 XC 3S1000, shown above, which had a million logic gates and 24 I/O pins as well as a 432k on-chip Block RAM for storage and 24 18x18 bit multipliers. As you can see it has two columns of twelve RAM blocks on each side, each one 18k in size, giving us 432k total RAM available for storage, as blocks can be cascaded together. Each block has dual ports but can also be used as single port RAM. They also have a synchronous clock input, clock enable, set/reset and write enable as well as the usual address and data lines. All this information was available through detail Xilinx application notes[1] which explains everything from the layout to how block RAM is instantiated through VHDL. This document also led me onto other sources to research the composition of FPGAs in general, as well as rereading my FPGA notes from my current and past courses.

The project board also had some external SDRAM, a CPLD, 4 dip switches, an 8 segment LED display, 2 push buttons and all the necessary ports for loading on projects and even displaying outputs on a monitor if needed.

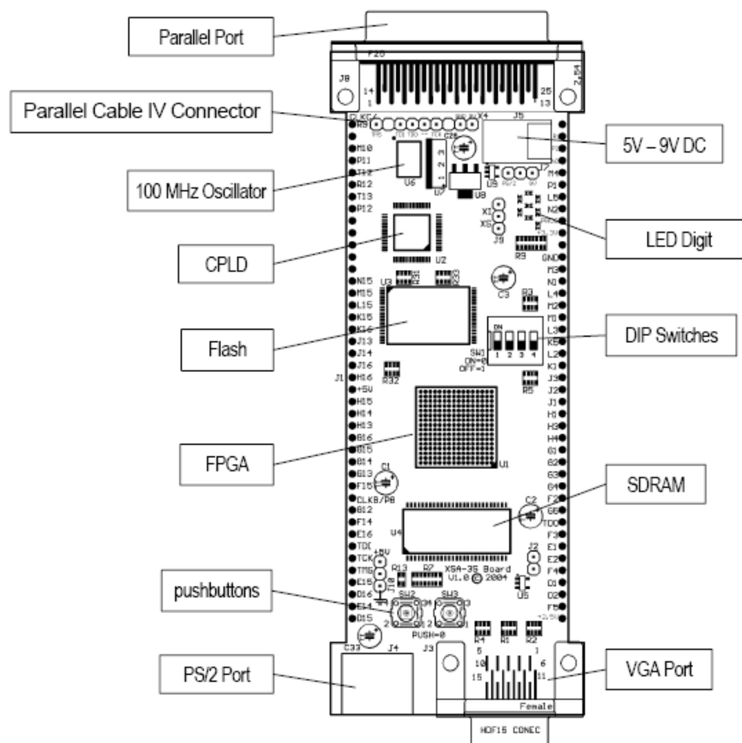


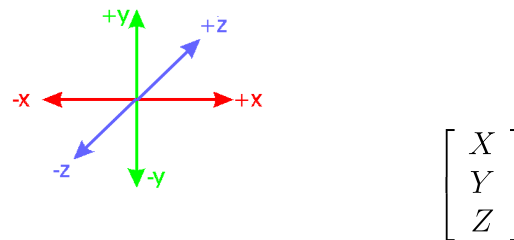
Figure 2.2: XSA-3S1000 Board

2.3 3D Graphics

3D graphics are images made up to represent our 3D world and its objects. Besides being used for games and other computer applications, 3D graphics are also very important in many other sectors with scientific, engineering and medical uses. One example is in the medical industry, where 3D graphics are a necessity in volume rendering of MRI brain scans. This is where a 3D model of a patient's brain is built up and displayed on a screen to help doctors discover problems without the need for surgery.

Each 3D image contains many 3 coordinate points which represent a point or vector in 3D space. These 3 coordinates represent the vectors location along each of the X, Y and Z axes which together make up 3D Space. This can be seen in the image below with the X axis representing the horizontal, length dimension, the Y

axis representing the vertical dimension, making area and the Z axis perpendicular to both of these, going directly towards the viewer and away from them, creating the third dimension, volume. Also below is a vector displayed as a matrix rather than the usual coordinate format of (X, Y, Z). All vectors can be displayed both ways and this makes it possible to transform the vector by multiplying it by one of the transform matrices, as follows in the next section.



$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Figure 2.3: Axes of 3D space [2] and a Vector in Matrix Form

A combination of a small number of vectors can create simple objects such as cubes and pyramids but in order to represent real world objects we need a lot more detail. This can be generated by NURBS modelling (drawing with a combination of curves) or Constructive Solid Geometry (CSG - where primitive shapes such as spheres, cones and cuboids are combined to make up the image) but both of these are usually converted to polygons for polygonal modelling. This is the most common form of modelling, where each vector forms a side of a polygon, usually a triangle but any polygon can be used depending on algorithms and hardware. As can be seen below in the pictures of the cow, these polygons combine to form a wire mesh frame of the image. If the number of polygons is increased and their size reduced, we get a much more detailed image as can be seen in the picture on the right. This also means we need an increase in the number of vectors and there can be millions of vectors in a fairly basic 3D scene, changing constantly. It is clear that fast, accurate transformations and an efficient way of manipulating these vectors is essential for the smooth running of any 3D graphic application.

After this wire mesh is constructed and all vector points have been transformed and modelled; lighting and shading takes place. There are many different, complex ways of doing these processes but the main idea is that each polygon is shaded or covered over with part of an image or texture so that the mesh is filled in and

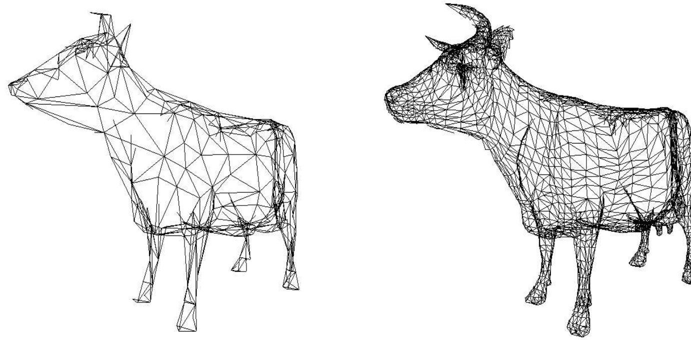


Figure 2.4: Wireframe Cows [3]

looks solid. These colours/textures are lightened or darkened depending on where the light source is in the scene, creating shadows where we would expect shadows in our 3D world. As can be seen below, the cow is simply coloured shades of grey with shadowing to the front and below the cow, indicating that the light source is above and behind the cow. Notice also that we only see the near side of the cow as the vectors and polygons behind are hidden. This is known as hidden line removal or depth buffering and is also performed at this stage. On computers, all these processes occur in different stages of the 3D Rendering Pipeline.

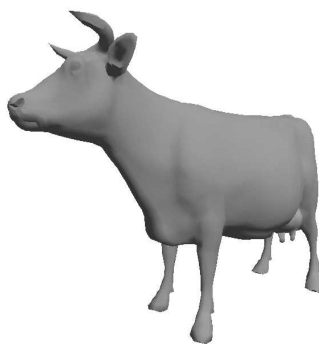


Figure 2.5: Shaded and Lighted Cow [3]

2.4 3D Graphics on Computers

2.4.1 3D Graphic Pipeline

3D graphics on computers are handled with the 3D Rendering Pipeline. This has three main stages, Application, Geometry and Rasterizer. In the application stage the scene is taken in and broken down, with some effects such as animation and collision detection occurring here. This outputs vectors to be dealt with in the geometry stage. Within the geometry stage there exists another pipeline, detailed below, which outputs the final scene to the rasterizer which adds colour and performs depth buffering.

2.4.2 3D Geometry Pipeline

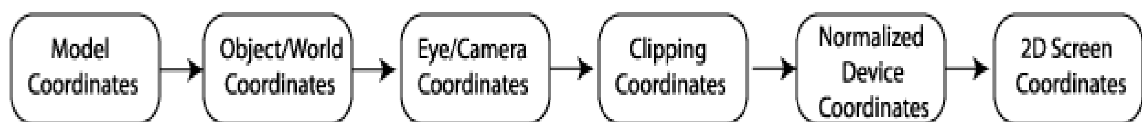


Figure 2.6: 3D Geometry Pipeline[4]

The geometry pipeline usually has six stages. The first stage receives the model coordinates from the Application stage. Model coordinates are the coordinates that define an object in object space, that is, usually centred on the origin with distances defined in relation to this. If you can imagine the cow above while the designer was drawing it in 3D design software, it was the sole inhabitant of the screen with its coordinates and size defined in object space. By making each object in model coordinates it means we can easily transform and duplicate that object separate to our scene and then translate it back in.

It is in the 3D World Coordinate stage where this scene is built and all objects are translated into their positions and their coordinates converted to reflect their position in the scene. This scene can be as big as it needs to be but it will be clipped down to fit onto the screen later as I will discuss below. Lighting sources and shading colours for the scene can also be calculated at this stage.

In the Eye/Camera Coordinates stage the 'camera' is transformed so that it is looking along the Z axis in a positive direction with the positive Y axis going vertically up and the positive X axis horizontally to the right. A viewing frustum is constructed to restrict the scene so that objects are not placed out of the final screen view. The frustum can be described as a truncated pyramid on its side (as seen below) with a back and front plane, at which the viewer/camera looks. Typical values for these planes are +1 for the front plane and +1000 for the back plane; both on the Z axis, but these can be set to any value depending on how you want the final view to appear. All objects in the scene that are to be seen in the final view must be translated inside this frustum.

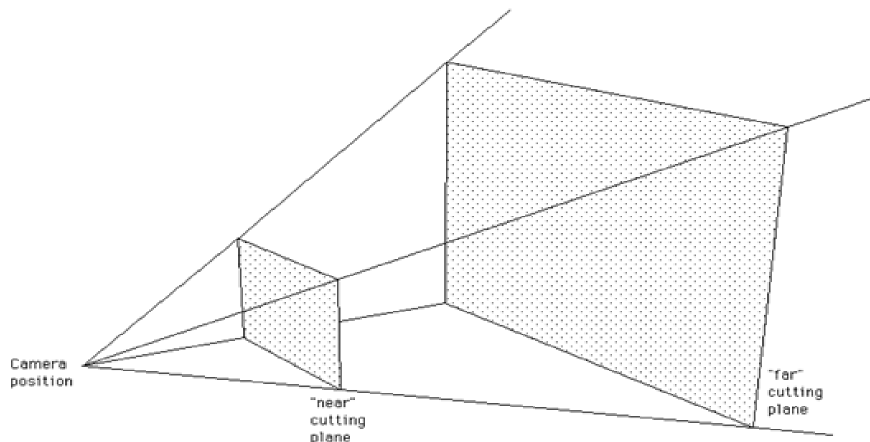


Figure 2.7: Viewing Frustum[5]

In the Clipping Coordinate stage, the scene is normalized, transformed and projected into a 2D view. Normalization is the rotation of objects or a whole scene so that its contents 'face' the camera, that is face the X and Y axes as described above. Other transformations such as scaling and translations can be performed to fit the desired scene into the final view. How the scene is viewed then depends on the type of projection used in this stage. There are two main types, orthographic and perspective projection, both of which return a 2D value for each vector, which I will discuss in more detail later.

In the Normalized Device coordinates stage, the first process is the conversion of homogenous coordinates back to 3D coordinates if homogenous W is not '1',

the reasons for which I describe later. Clipping is performed next in which objects that are not to be included in the screen view are clipped away if they are outside the bounds of the front and back planes of the frustum. If the front clipping plane slices the minority of an object out of view (for example, if the cow's head were to contain a Z coordinate less than +1), the object would be translated into the frustum and other objects adjusted if necessary. If the back clipping plane slices an object out of view it could either be truncated or clipped away completely from the view.

The final stage in the geometry pipeline is the 2D screen coordinate stage. Here the 2D coordinates are converted into pixels to be drawn to the screen. Hidden line/surface removal can be performed here or later in the rasterizer stage. Anti-Aliasing can also be performed here which is the smoothing of jagged edges by filtering the image.

2.5 3D Graphic Transformations

2.5.1 Transformation Matrices

My project was based mainly on the clipping coordinate stage but as you can see transformations span many stages of the 3D rendering pipeline and are therefore an extremely important and complex component. Transformations are the rotation, scaling and translation of the each vector that makes up an image. Each type of transformation is represented by a transformation matrix. This is a matrix which is made up from the identity matrix but with some alterations to perform the necessary transformation. As you know a 3x3 identity matrix, when multiplied by any 3x1 matrix (for example our vector in matrix form), results in the original matrix, as shown below. The changes to this matrix give us the ability to transform vectors in different ways. The transformation matrix is then multiplied by the input vector to produce a transformed result.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Figure 2.8: Identity Matrix

2.5.2 Matrix Multiplication

Matrices are multiplied by a combination of multiplication and addition. First the X part of the input vector is multiplied by each element of the first column of the transformation matrix, namely A1, B1 and C1. Y is then multiplied by the second column and Z by the third. The results of these multiplications are then added as shown below to give the transformed result.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \times \begin{bmatrix} A1 & A2 & A3 \\ B1 & B2 & B3 \\ C1 & C2 & C3 \end{bmatrix} = \begin{bmatrix} \text{(X * A1)+(Y * A2)+(Z * A3)} \\ \text{(X * B1)+(Y * B2)+(Z * B3)} \\ \text{(X * C1)+(Y * C2)+(Z * C3)} \end{bmatrix}$$

Figure 2.9: Matrix Multiplication

2.5.3 Scaling Matrix

When you want to scale a vector by a value, the scale matrix (below) is filled with the values needed to scale the X, Y and Z part of the vector by, namely Sx, Sy and Sz. As you can see, these replace the digit '1' in each row/column of the identity matrix, therefore multiplying our input vector values by the values stored in this matrix. This has the effect of increasing our X, Y and Z by the degree of whatever we store in Sx, Sy and Sz respectively.

$$\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & Sz \end{bmatrix}$$

Figure 2.10: 3x3 Scale Matrix

2.5.4 Rotation Matrices

To rotate a vector, you must first choose which axis you wish to rotate around. For each axis there exists a different rotation matrix, each calculated using trigonometry due to the circular nature of rotations around an axis. The signs of the elements of the matrix depend on which way you want your rotations to occur, that is in a clockwise or anti-clockwise direction around each axis. The matrices below use the anti-clockwise or right handed system. This can be easily understood if you take your right hand, point your thumb in the positive direction of the axis you wish to rotate around and the direction your fingers curl is the direction of rotation.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & Cos\ \alpha & -Sin\ \alpha \\ 0 & Sin\ \alpha & Cos\ \alpha \end{bmatrix} \begin{bmatrix} Cos\ \alpha & 0 & Sin\ \alpha \\ 0 & 1 & 0 \\ -Sin\ \alpha & 0 & Cos\ \alpha \end{bmatrix} \begin{bmatrix} Cos\ \alpha & -Sin\ \alpha & 0 \\ Sin\ \alpha & Cos\ \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2.11: 3x3 X,Y and Z axis Rotation Matrices

2.5.5 Homogenous Co-ordinates

All the above matrices can be multiplied by the input vector to produce a scaled or rotated output vector. The problem arises when we want to translate a vector or component of a vector. With translations we want to add to the current value of X, Y or Z, not multiply. In order to do this, we need to introduce a fourth row/column to our transformation matrix, the homogeneous row/column. This gives us the ability to put a value we want to add to the input vector in Tx, Ty and Tz, as seen below. But in order for this to work out we need to also add a homogeneous coordinate to our input vector. We call this coordinate W and usually set its value

to '1' so that when we multiply our input vector by our translation matrix; Tx, Ty and Tz get multiplied by '1' (as W is multiplied by the fourth column) and are then added to the input coordinate (which is kept the same as the rest of the matrix is the identity).

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

Figure 2.12: 4x4 Translation Matrix and Vector matrix with Homogeneous W

The homogeneous coordinate W can actually be set to any value once X, Y and Z are scaled up accordingly so that x/w, y/w and z/w bring them back to their original values. This is necessary as we need to reduce our transformed vectors back to 3 coordinates (X, Y, Z) again before we can display them on the screen. It is therefore simplest to keep W set to one.

Since we have extended our transformation matrix from a 3x3 to a 4x4 matrix, we must add another row and column to the scale and rotations matrices. In order to not change their functionality, we simply add a line from the identity matrix which keeps homogeneous W set to whatever value it is. This gives us new matrices for scaling and rotation as shown below.

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.13: 4x4 Scale Matrix with Homogeneous W

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos\ \alpha & -Sin\ \alpha & 0 \\ 0 & Sin\ \alpha & Cos\ \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Cos\ \alpha & 0 & Sin\ \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -Sin\ \alpha & 0 & Cos\ \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Cos\ \alpha & -Sin\ \alpha & 0 & 0 \\ Sin\ \alpha & Cos\ \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.14: 4x4 Rotation Matrices with Homogeneous W

2.6 3D Graphic Projections

Another part of the clipping coordinate stage in the geometry pipeline, and my project, is the projection of the 3D transformed vectors onto a 2D screen. As you know, screens can only show images in two dimensions, length and area. If we want our final view to appear 2D we can use an Orthographic projection. Alternatively we can use perspective projection techniques to make these 2D images appear to have a third dimension, volume.

2.6.1 Orthographic Projection

Orthographic projection, also known as parallel projection, projects straight, parallel lines out from each corner of the image and draws the image onto what can be imagined as a glass box surrounding the object (see image below). This can be done in either first or third angle ways. First angle is when facing one side of an object, its image is projected in the direction of view onto the glass wall behind the object. Third angle is the opposite, with the side of the object you are viewing being projected towards the viewer, on the glass wall between the viewer and the object. This is called third angle projection as it needs two 'glass box' rotations to get back to first angle. In my project, this glass box is the equivalent to rotating the object around the different axes and taking an orthographic projection at each side to build up a complete orthographic projection of the object. A third angle projection is shown below and as you can see, when we open up the box, we have a perfect scaled drawing of each side of the object. It is this ability to keep perfect measurements that makes orthographic projection so important for engineering and other technical uses. This example used a 1:1 scale proportion but by reducing (or increasing) this we can draw any size object to proportion so that

exact measurements can be extracted. This proves indispensable to any industry which requires blueprints or plans, from massive building construction projects to minuscule chip design ones.

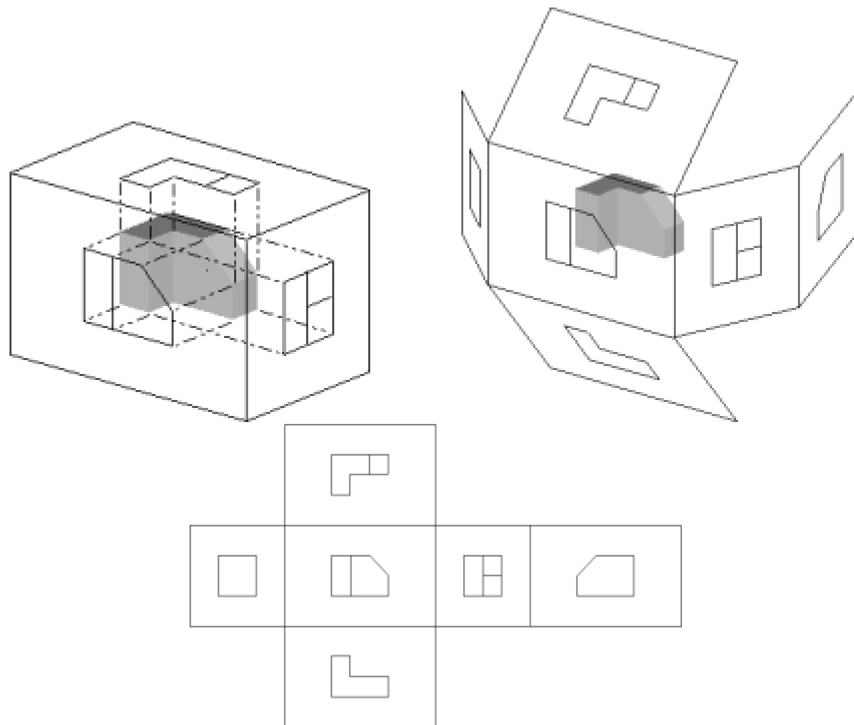


Figure 2.15: Orthographic Projection
[9]

Orthographic projection is mathematically very trivial. In order to perform this parallel line projection, all that is needed is to rotate the object so it is normalized to the axes, facing full on side, front or back. Then both Z and W coordinates of each vector are discarded and the X and Y coordinates are projected and their corresponding lines drawn on the screen.

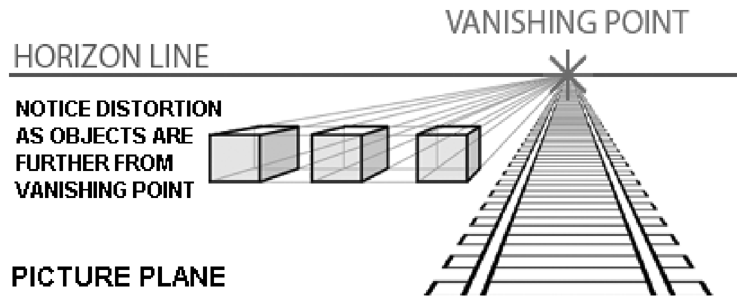
2.6.2 Perspective Projection

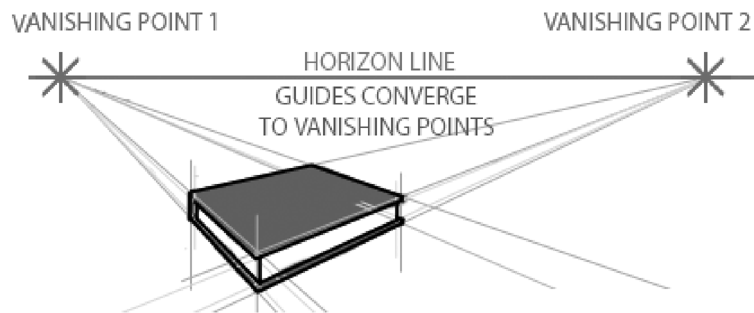
Perspective projection differs from orthographic projection in that its emphasis is on making the image appear to have depth, thereby mimicking our view of the world, rather than preserving scale. This method of projection goes back centuries

and was used to depict more realistice images, first seen in paintings by Filippo Brunelleschi in the early 15th century. It can also be seen in famous renaissance paintings such as Leonardo Da Vinci's 'The Last Supper'.

Perspective projection tries to mimic human vision, which uses both our eyes to calculate depth perception in our three dimensional world. The problem is the viewing frustum only offers one viewpoint, much like a camera. Therefore perspective projection emulates our view based on the principle of vanishing points. If you can imagine yourself standing on a railroad tracks and looking along them, the tracks seem to converge as they reach the horizon. If you were to place a sheet of glass in front of you and draw the scene, this is what perspective projection aims to do. To accomplish this, it projects the points and lines of the objects towards this vanishing point, making further away objects smaller, just like when we view objects in our world. This depth of objects is also crucial for hidden line/surface removal to be preformed accurately, so that objects further back are obscured by objects in front.

In this train track example there is only one vanishing point on the horizon, which is sufficient for some scenes but can lead to distortion of objects further away from the vanishing point (see the left cube in the one-point perspective picture). To add more detailed perspective, more vanishing points are added which gives an added sense of realism.





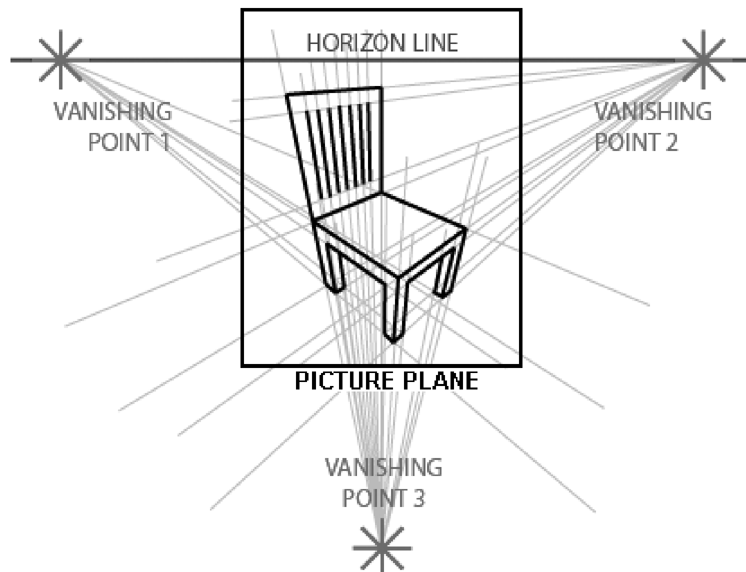


Figure 2.16: Vanishing Points in Perspective Projection [6]

This method of projection is based on a projection matrix and is much more mathematically complex than orthographic projection. This is because the projection stage can be combined with normalization to the X and Y axes, translation of the scene to the origin and scaling and clipping the scene to fit into the frustum, all which can also be handled by other stages of the 3D Geometry Pipeline. All these steps together over complicate the perspective matrix and there are many different methods of constructing the final matrix, for example the Open GL method.

Failing to find a dedicated book on the subject I opted to concentrate on the simple perspective matrix which featured most, albeit in different forms, on many websites and is a basic perspective matrix without any extra steps. I chose the matrix from the most reputable source, American and Canadian university' notes, which is shown below and is very similar to the matrix specified in our own colleges Computer Vision course notes.[10, 7, 11] As you can see it is almost identical to the identity matrix except homogenous W is now going to work out as Z/d, where d is the distance from the camera/viewpoint to the front plane of the frustum.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Figure 2.17: Perspective Projection Matrix

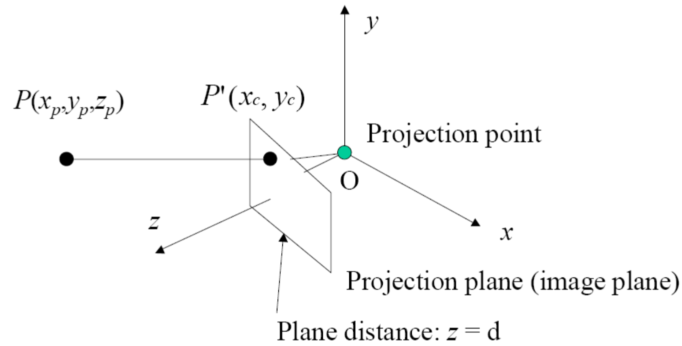


Figure 2.18: Perspective Projection[7]

This matrix is calculated in the following way: If we have a line, OP, going from the origin (where the camera/viewpoint is) through the image plane (the front plane of the frustum) to a point in 3D space, P, we can calculate where the image plane intersects that line at P'. To do this we use the parametric equation of a line formula and get $x = tXp$, $y = tYp$ and $z = tZp$. We can then put $z = tZp$ into the plane equation $Z = d$ and work out that $t = d/Zp$. We can then calculate that the final point P' as

$$(\frac{d}{z_p}x_p, \frac{d}{z_p}y_p, d) \tag{2.1}$$

We can confirm this by multiplying an input vector (X, Y, Z, W) by the perspective matrix and dividing across by homogenous W, as shown below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \text{1/d} & 0 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} X & 0 & 0 & 0 \\ 0 & Y & 0 & 0 \\ 0 & 0 & Z & 0 \\ 0 & 0 & \text{Z/d} & 0 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ \text{Z/d} \end{bmatrix}$$

Figure 2.19: Multiplying the Perspective Projection Matrix by an input Vector

$$\begin{bmatrix} \mathrm{X/(Z/d)} \\ \mathrm{Y/(Z/d)} \\ \mathrm{Z/(Z/d)} \\ \mathrm{(Z/d)/(Z/d)} \end{bmatrix} = \begin{bmatrix} \mathrm{d/Z}\times\mathrm{X} \\ \mathrm{d/Z}\times\mathrm{Y} \\ d \\ 1 \end{bmatrix}$$

Figure 2.20: Dividing across by Homogenous W

2.7 3D Graphic Hardware

2.7.1 Graphics Processing Unit

In most modern computers, dedicated graphics cards perform most of these complex calculations of the 3D graphics pipeline. In the past all these computations would have been performed on the CPU along with all other general purpose calculations. As 3D graphics advanced, these calculations would have taken up mass amounts of CPU timeslots and other applications would have suffered and their performance deteriorated. As a result, graphics cards were introduced into modern computers to work alongside general purpose CPUs, taking the strain of the high volume of matrix and vector calculations needed for 3D graphics. These cards each have their own microprocessor called the Graphics Processing Unit, which handles these calculations, including geometry and mapping between coordinate systems. These GPUs receive and deliver results over high speed buses to the RAM and CPU. These buses between the GPU and RAM are necessary as although the graphics cards have some on-board RAM, this is usually not large enough for the millions of calculations performed and delivered back to the CPU per second, so the computers RAM is used too.

These GPUs were so fast at performing these matrix and vector calculations that people began using GPUs for other, non-graphical purposes; such as generating stereo images - 3D images made from two photos taken at slightly different angles viewed through 3D glasses or with eye focusing techniques.

It was about the same time that people began to look at alternatives for hardwired ASICs on which older GPUs were based. Companies began introducing limited programmability to GPUs in the form of Application Specific Processors (ASPs) which basically could be used to run specialized code set up to improve

the processors ability depending on how it was being used. Although this was a step forward for GPUs, the programmability is quite limited and offered nothing like the freedom of fully reconfigurable hardware such as FPGAs.

2.7.2 FPGAs as an Alternative

As I have already stated, FPGAs are fully reconfigurable hardware devices that are simple to program and cheap to buy. Their configurability is so versatile it can emulate any type of hardware, even complex GPUs. This totally opened up graphic processing to anyone who wanted to attempt to implement the many new ideas and algorithms emerging all the time. The fact that FPGAs are dynamically reconfigurable added even more parallelism. This meant different bit-patterns could be loaded in from the on board RAM, totally changing the chips configuration to tailor it for its current application, and all this happening after the chip has been programmed, while it is working away inside a computer.

Although FPGAs are cheap in small numbers, ASICs are actually less expensive for large scale production but since they are quite similar, FPGA designs can be ported to ASICs. This is why FPGAs are a perfect hardware for people to try out their designs on, reloading their design if bugs are found, until a perfect design can be mass produced through ASIC vendors. FPGAs have even helped spawn a whole new type of 'soft' hardware where code written in hardware description languages is shared under an open license on websites such as opencores.org, where I found some useful modules for my project. One group of people have even gone as far as to create an Open Graphics Project[12], where they aim to produce an open source GPU on an FPGA, aimed at many of the lesser known operation systems and using open source drivers.

As you can see, FPGAs have some advantages as a possible alternative for GPUs but they still have some barriers preventing this. One such barrier is speed. FPGAs are currently slower than ASICs, which is a huge drawback when it comes to graphics as speed is a necessity. Another advantage ASICs have over FPGAs is lower power consumption, which is extremely important in mobile devices. Another disadvantage with FPGAs is the area required for reconfiguration, which uses up valuable storage space. However, the rate of improvement in both speed

and size of FPGAs currently exceeds the rate of improvement in both GPUs and ASICs, so FPGAs are sure to improve in the near future , in accordance with Moore's Law.

2.7.3 FPGAs as an Assistance

It is for this reason that although FPGAs may never fully replace dedicated graphic processing units, I can see them collaborating with both the GPU and CPU, providing relief work for some of the computations which require more parallelism than the GPU can offer or just to clear up some more timeslots on the CPU and GPU. Other advantages an FPGA could offer are the ability to reconfigure parts of the GPU for bug fixes after its release or even to tune the GPU more towards its current function.

One idea for FPGAs aiding in graphic processing is the current research in Trinity College into Tile-Based Rendering, which I hope my project might possibly aid. This basically involves a cluster of graphics cards, each with their own FPGA available on board, connected over a very high speed Scalable Coherent Interface (SCI) to a central computer. On this main computer the scene, object or application (for non-graphical purposes) is broken up into a series of tiles through software which are then sent to their own individual graphic card, processed and returned to be collated again on the main computer. By this method of parallelism, the research team aim to process at a speed unmatched by any graphics card in a single PC. It is for this reason that the somewhat overlooked FPGA is now being tested for different ways to utilise it to help make the GPU and CPU cluster combination more efficient. My project, handling transformation and projections, is just one of many possible ways a cheap, reprogrammable device can assist in graphic processing. This assistance can also be applied outside of graphic applications, to many other computing tasks.

2.8 IEEE Single Precision Floating Point Numbers

Floating point is a way of representing rational numbers in binary. Each number usually has a sign bit, an exponent and a mantissa. This is the equivalent of scien-

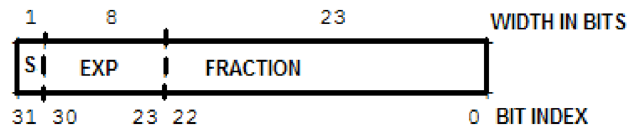


Figure 2.21: IEEE Single Precision Floating Point Numbers

tific notation in the decimal system, except that in binary the base of the exponent is two, not ten.

The Institute of Electrical and Electronics Engineers are a group responsible for setting standards in all aspects of electrical engineering, and they set the standard for binary floating-point arithmetic to four formats in the IEEE 754 Standard. The most commonly used format is single precision, which uses 32 bit binary numbers. This is the format I chose for my project as it was one I was familiar with and gave a large and detailed number range.

IEEE Single Precision floating point is divided up into one sign bit, s, eight exponent bits, exp, and 23 mantissa bits, as shown above. There is also a hidden bit between the mantissa and exponent and this is always set to 1, giving an extra bit of precision but also requiring a special case to represent zero (as seen below), as true zero cannot be defined. The sign bit represents the sign of the number and is set to '1' for negative or '0' for positive. The mantissa field represents the integer or fixed point part of the number. The exponent bits are the index that the base is raised by. This field can represent numbers in the range 0 to 255. But as we want our exponent to represent numbers in the range -126 to +127, we apply a bias to the exponent field. What this means is, when extracting a number from floating point, we subtract 127 from the exponent field before raising the base, two, to the power of the value in the exponent field. This is then multiplied by the mantissa to get the result. Put more simply, a floating number fpnum is calculated by the formula

$$fpnum = mantissa \text{ x } base^{exp} \tag{2.2}$$

This describes the most common type of floating point number, normalized numbers. An example of such a conversion follows.

To encode the decimal value +30.625 to floating point representation we first convert to binary. This yields 100011.101. (Decimal places are converted by the most significant bit after the decimal point representing 0.5, the next most significant half of this, giving 2.5 and the subsequent bits half their former bit).

This binary number is then shifted to the right until there is a single 1 on the left hand side of the decimal point (which becomes the hidden bit), yielding 1.00011101. The hidden bit is removed and the rest padded with zeros to 23 bits to become the mantissa.

Since we shifted right by five places, the exponent is five. But this must be biased with +127 giving +132, or 10000100 in binary. This is the value of the exponent field and since the number is positive, the sign is set to '0' giving:

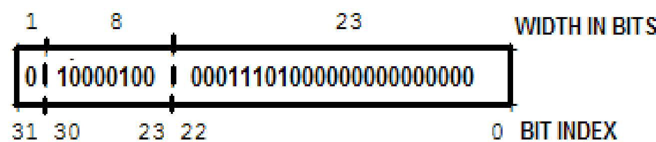


Figure 2.22: IEEE Single Precision Floating Point Example

The IEEE standard also represents other types of floating point numbers including denormalized numbers, infinity (positive and negative), zero (positive and negative) and invalid results called Not a Number which result from invalid operations such as a multiply by infinity. How these are represented is shown in the table below.

Class	Exp	Fraction
Zero	0	0
Denormalized Numbers	0	non-zero
Normalized Numbers	1-254	any
Infinities	255	0
NaN (Not a Number)	255	non-zero

Table 2.1: IEEE Single Precision Floating Point Representations

Chapter 3

Design and Implementation

In this section I will discuss my different approaches to the project, and step through the design, implementation and testing of each one as well as what I learned at each stage.

3.1 Binary Multiplier

3.1.1 Design

When I began the implementation stage I decided to take a bottom-up design approach, making and testing each element of the project individually and gradually integrating these designs together, testing after each addition, until I had a complete design. I had some loose ideas on what I would require in my project but knew a multiplier was essential so decided to start with that. Throughout my college course I had used Mano and Kime's 'Logic and Computer Design Fundamentals'[13] to accompany my hardware courses and this was the first resource I turned to. This provided me with a Binary Multiplier design I decided to implement in VHDL.

3.1.2 Implementation

As you can see, the multiplier itself has many components. I began coding each one individually, testing that each part worked with basic input values before go-

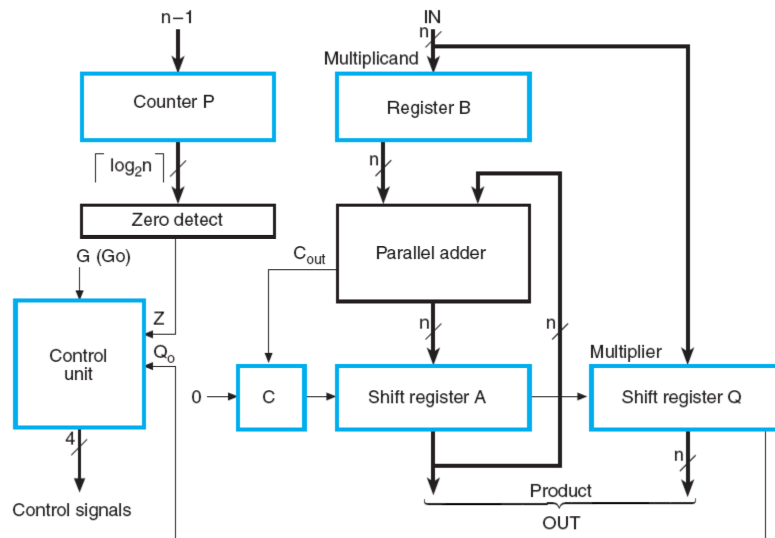


Figure 3.1: Mano and Kime's Binary Multiplier Block Diagram
[13]

ing onto the next. I decided to keep things simple at first, using only eight bit numbers. I coded some simple D flip flops for registers, a counter module to count down to zero from any value loaded in, a zero detect to identify when count had reached zero, a ripple carry adder incorporating full adders and a shift register with parallel load.

Although these components are quite common, I had not coded VHDL in a couple of years and although they helped me get reacquainted with coding, I did found them quite time consuming. After doing these components I noticed that it would take a lot of time to get this binary multiplier fully working as I still had the most difficult part, the control unit, to do and even if I did get this design working, it would only be useful while I was using eight bit values as I would require a totally different multiplier when I switched to floating point. For this reason I decided to move on. Before I did, I tried typing out the code for a state machine binary multiplier directly from Mano and Kime's book but this failed to synthesize and proving time consuming, I decided to simply use VHDL's built in multiplication and addition notation.

3.1.3 Conclusion

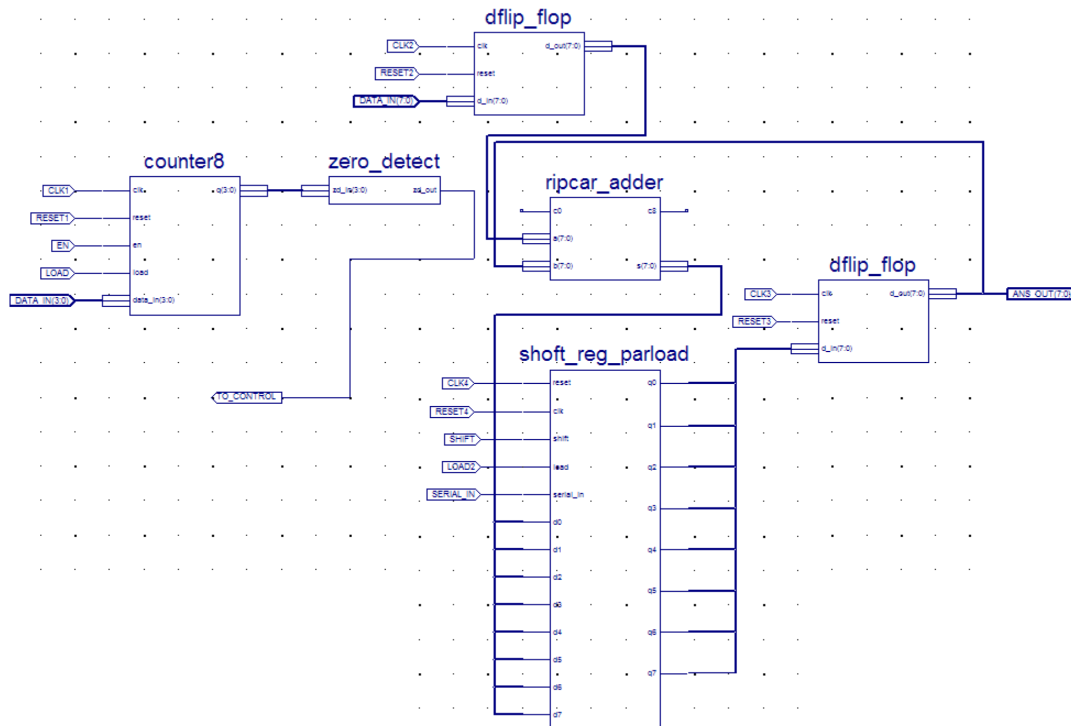


Figure 3.2: Components of My Binary Multiplier

Overall I suceeded in getting several components working individually, which helped ease me back into coding and the hardware state of mind, but as I did not finish the whole design due to time constrictions, I could not test it together. A block diagram of the parts that I coded is shown above.

3.2 Behavioural Design

3.2.1 Design

Now that I could perform simple multiplication I needed to decide upon a design. I decided to implement a design to generate transforms and orthographic projection without matrix multiplication to evaluate the result I could obtain this way. I also

expanded the size of my input values to nine bits so I could use the most significant bit as a sign bit. I came up with a basic design as shown below.

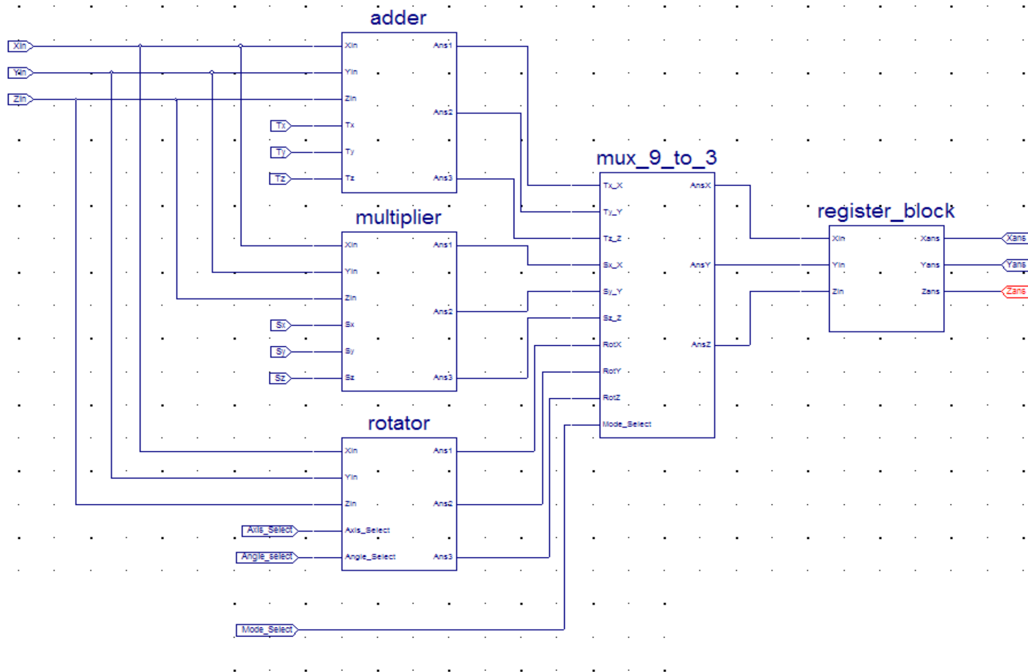


Figure 3.3: Behavioural Design

3.2.2 Implementation

As you can see, I split my design up into four components. The two smallest components were for addition and multiplication and used VHDL's '+' and '*' operators to add and multiply inputs, taking into account the sign bit of the inputs and assigning polarity to the outputs accordingly. I made these components separate to the top file that encapsulated them (not shown) so that I could insert a floating point multiplier and adder in their place later in my design. This encapsulating file read in the X,Y and Z coordinates, the values to scale them by, Sx, Sy and Sz and the values to translate them by, Tx, Ty and Tz. I used port maps to pass these scale values and inputs through three instances of my multiplier component. I also passed the translate values and inputs through three instances of my adder.

Another component of my design calculated simple rotations of 90, 180, 270 or 360 degrees. This basically changed the sign bit of the inputs depending on which axis the inputs were rotated around and which of the four possible angles were selected. For example if a point (2,2,2) is rotated 90 degrees (anti-clockwise) around the positive Z axis, the Y part becomes negative while the X and Z parts remain positive, leaving the result (2,-2,2).

I later tried to add more rotation degrees such as 15, 30 and 45 degrees but this involved calculating cosines and sins which worked out as floating point numbers so I decided to leave these rotations until I had expanded my design to floating point.

As this design was to be a simple behavioural design, each transform was performed separately on the input vector. This meant there were nine separate outputs, 3 scaled, 3 translated and 3 rotated. To make my design neater I made a multiplexer component with a select line so although all three transforms were performed every time, only whatever result the user wanted would be output. These output coordinates where then latched with some registers so that they were all outputted simultaneously. Taking just the X and Y outputs then gave me the result in orthographic projection.

3.2.3 Conclusion

I tested this complete design with test bench waveforms, which is a simple method of testing. Using this method you can select input values for each of the inputs, simulate your design as if it were on hardware, and the outputs of your design are there for you to read off the waveform. I discovered that this design worked perfectly for the small range of test values I inputted and concluded that it worked sufficiently for a behavioural design. Unfortunately, I continued work from this design for my next stage, failing to save a distinct copy of the source code and test bench waveforms for inclusion with this paper, although I believe I could now easily code this design in a few days since I have done it once before.

Although this design was only a behavioural design, it helped me to get the aims of my project clear in my head so that I knew what my final design should incorporate and accomplish. This design was coded at the same time a lot of

my research was still being done so it provided a practical means of viewing the theory behind 3D graphic transformations. Also, the actual process of deciding on a design of my own got me thinking about the architecture of the hardware my design was going to be loaded on, as well as giving me the chance to practice coding of my own design rather than just coding components of a pre-existing design

3.3 Memory Load Design

3.3.1 Design

When I was satisfied with my behavioural design I began working on the problem of introducing matrices into my design. Before I built a component to input the matrices I worked at making a module to multiply matrices. I comfirmed the matrix multiplication algorithim and considered alternative methods to implement this in VHDL. Looking back now this might appear like a trivial step but at the time I was actually working through a few different methods, evaluating the pros and cons of each. I came across an application note from Xilinx regarding matrix multiplication on a Virtex-II FPGA[8] which included a block diagram of the circuit, as shown below.

I considered using this design as a basis for my project, extending it to handle 4x4 matrices which would be stored in registers and selected by multiplexers. I would also have had to extend each element of the design to 32-bit floating point in order to work with a high accuracy. I decided this was not the best way to approach my problem but decided to keep the idea of three separate stages, the first being a matrix builder stage to hold and load the matrices.

The second stage would be a block of sixteen multipliers, rather than a single multiplier with counters to control the flow of inputs, as this could handle the multiplication of all elements of the matrix at once, thus speeding up the design.

The third stage would contain a block of twelve adders to complete the multiplication of the matrix, the first eight adders to calculate the temporary sum of each half of the four multiplication results that make up each transformed coordinate and the last four to add these temporary results, giving the final transformed

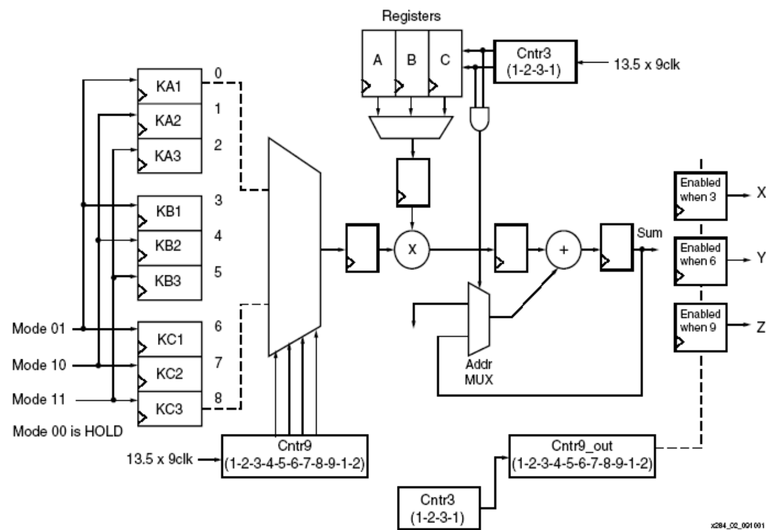


Figure 3.4: Xilinx 3x3 Matrix Multiplier Diagram[8]

result. It was necessary to do this in two halves as each adder could only calculate the sum of its two inputs.

tmp1 + tmp2 = ans1

$$\begin{bmatrix} (X * A1)+(Y * A2)+(Z * A3)+(W * A4) \\ (X * B1)+(Y * B2)+(Z * B3)+(W * B4) \\ (X * C1)+(Y * C2)+(Z * C3)+(W * C4) \\ (X * D1)+(Y * D2)+(Z * D3)+(W * D4) \end{bmatrix} \quad \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

Figure 3.5: Two Stage Addition

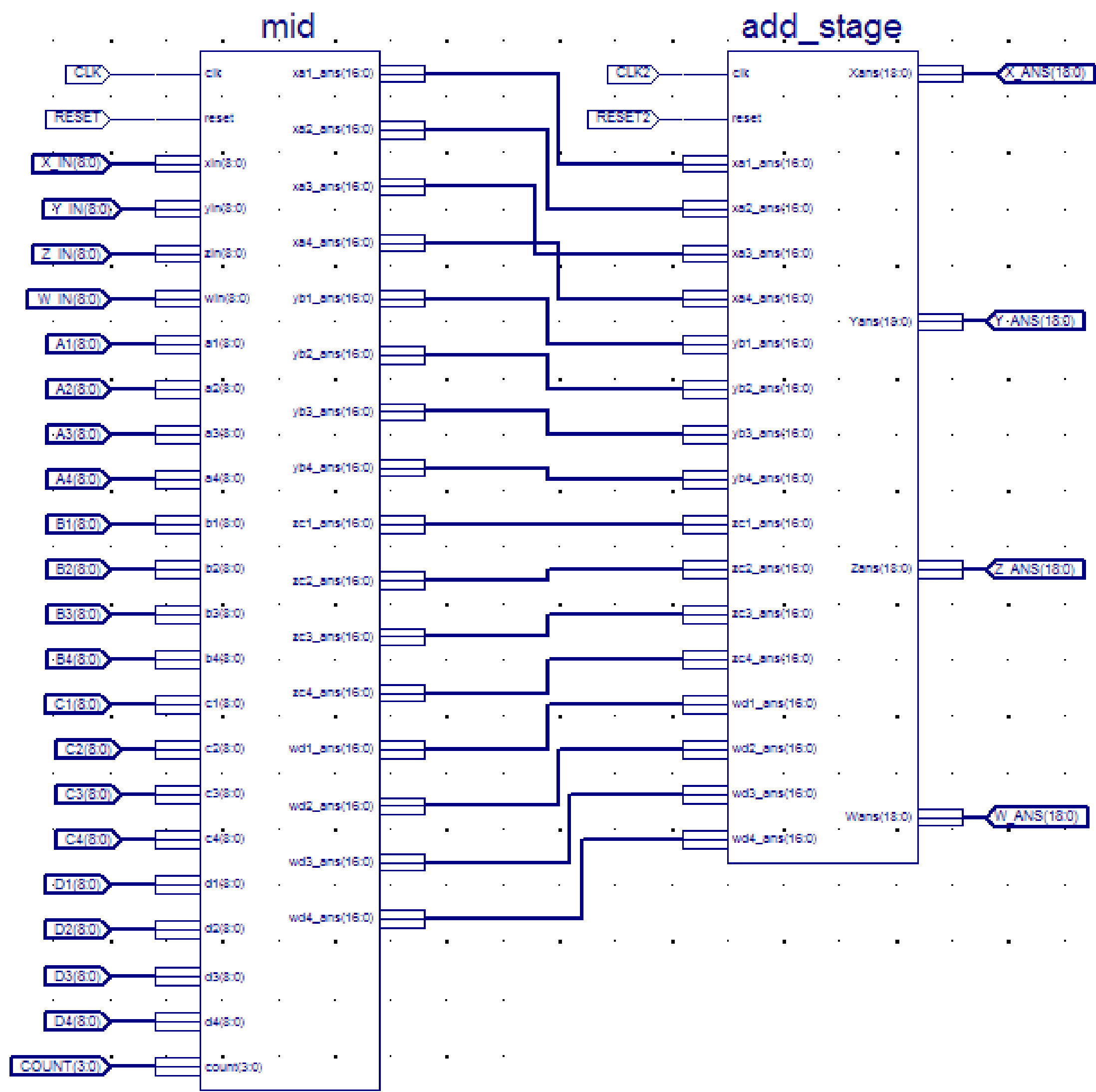


Figure 3.6: Multiplication and Addition Stages

3.3.2 Implementation

I constructed these multiplier and adder blocks by instantiating the multiplier and adder components from my behavioural design. I tested these individually and then together with test bench waveforms before moving on. I next had to decide on how to implement my matrix builder. Although I could just input them directly or store them in registers, I decided that neither of these were the most efficient way to implement my idea. After talking to another student about storage, the FPGAs block RAM was mentioned and I decided to look into this and began the research into the Spartan 3 block RAM as detailed in the hardware section.

It was during this research that I realised that the matrices changed depending on the inputs and so although I could store some parts in RAM, I would be better off building the matrices dynamically, depending on inputs.

It was also at this stage of my project that I decided to combine the transformation matrices into one before passing them to the multiplier so as to complete the three transforms in one step, thus vastly improving the speed between input and output of my design. I also decided it was time to expand my design to floating point and this led onto my final design.

3.3.3 Conclusion

As you can see, I learned a lot during this stage of my project, about FPGAs and their composition but also about my design. Although my idea changed a few times, I still had my basic pipelined design worked out and just needed to build around this.

3.4 Final Design

3.4.1 Design

This design built on my previous idea of a three stage model: Matrix Builder, Multiplier and Adder. It also had a separate Rotation component to calculate the cosine and sine of the input angle to avoid overloading the matrix builder stage.

My design was later extended to include my Projection stage which selected and implemented the projection type.

3.4.2 Implementation

Multiplication and Addition Stages

My first step in my final design was to extend my multiplication and addition stages to floating point. For this I needed to find suitable floating point unit (FPU) modules as to code my own to a high enough standard would be a whole project in itself. Without knowledge of the open source 'soft' hardware community at opencores, I searched the internet for FPUs, finding different designs and trying them out, each of them failing to compile or not being suitable for FPGA use, which was a necessity for my ultimate goal.

I finally came across The Howard University RARE project[14] and its FPU source code. This was the Remote Adaptive computing REsource project which aimed to create a fully optimized piece of hardware, incorporating Xilinx 4044 XL FPGAs, loaded with fully optimized VHDL FPU code for performing a whole range of complex tasks such as matrix multiplication and image processing, ten times faster than a high end computer could. They then planned to make this hardware available for people to use over the internet, sending their data to be processed on the optimized hardware and getting the results sent back. All this led me to believe that the FPU must be very efficient and so I chose the adder, multiplier and divider from here to instantiate in my project.

I used the multiplier and adders to extend my multiplication and addition stages to utilize floating point numbers and tested these. I also expanded and improved my D flip flop so that it now handled 32 bit floating point numbers and instantiated them in my multiplication and addition stages so that they registered the inputs before calling the multipliers/adders.

Matrix Builder

The next stage I worked on was the matrix builder stage. As I had seen both rotation and translation matrices and scale and translation matrices combined in-

dividually, I assumed that I could combine all three into one dynamically built matrix. This meant that I could then have a matrix that looked like the one below if I was to rotate around the Z axis, scale and translate all at once.

$$\begin{bmatrix} Cos\ \alpha \times Sx & -Sin\ \alpha & 0 & Tx \\ Sin\ \alpha & Cos\ \alpha \times Sy & 0 & Ty \\ 0 & 0 & Sz & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.7: 4x4 Transform Matrix for Translation, Scaling and Rotation around the Z Axis

I decided not to make this stage too complex by making a separate Rotation component to calculate the cosine and sine of the angle to be rotated by, α, which I will talk about later.

I implemented my matrix builder by having a mode selector called RVST, standing for Rotation Version Scale Translate, to determine the composition of the combined transform matrix. I used a case statement to determine what each element of the output matrix would be depending on this mode.

For example if the RVST mode was 0011, this would mean that no rotations were to be performed, only a scale and translate and the inputs Sx, Sy, Sz and Tx, Ty, Tz were sent to the outputs of the matrix builder in their proper positions.

The rotation version, or RV, was two bits, giving four modes: 00 for no rotations, 01 for rotation of X axis, 10 for Y and 11 for Z. If any of these rotation modes were set, the output was the relevant rotation matrix.

If both rotation and translate were set (i.e. RVST = 1x01, where x is a don't care value), the output matrix was just the relevant rotation matrix with the translates added to the homogenous column.

The problem was when I wanted to combine a rotation and a scale (i.e. RVST = 1x1x). Some of the elements of the final transform matrix would now share the same position. To solve this I introduced three multipliers to the matrix builder component, calculating the product of $Cos\alpha$ (inputted from the rotation component) and Sx, Sy and Sz separately, which shared matrix positions depending on the axis to be rotated around. This product then replaced the particular element of the matrix when it was required.

Rotation Component

As I had done some very simple rotations in my behavioural design, I expected my full rotation component to be quite straightforward when I expanded to floating point. I certainly did not anticipate that this one seemingly small task would cause the most problems for my final design. As with floating point units, cosine and sine calculations in VHDL are substantial tasks in themselves so I decided to instantiate an open source model into my project to complete this task. My first step was obtaining a VHDL mathpack, which had methods to complete many mathematical functions, including cosine and sine generation. My only problem was that these packages used real numbers and I needed floating point. I then embarked on a journey to discover how, if possible, I could convert between these two number systems, as VHDL had some conversion commands such as standard logic vector to integer converters. After trying to utilise a real-to-standard logic package I discovered that real numbers are not compatible with Xilinx synthesis tools and I would have to find another way to calculate my cosine and sine.

I next discovered some floating point packages in development by the Electronic Design Automation group with the aim to be included in future IEEE standard releases[15]. Their packages support 16, 32, 64 and 128 bit numbers but unfortunately, when I attempted to use these packages they needed to be added to the IEEE library and ModelSim could not add them successfully as there were still bugs in the code. Although they were not much use to me in this project, they are very useful packages and I would like to see them implemented in future IEEE standard releases.

I then consulted the final year project forum, where our supervisor and postgraduates helped us to solve problems, and was pointed in the direction of the CORDIC algorithm (COordinate Rotation DIgital Computer), a method of calculating cosine, sine and rotations using just shifts and addition, making it very efficient for reconfigurable hardware. There are a couple of different forms currently in development in the opencores community and I tried each one of these designs but unfortunately, each had a fatal flaw that would have taken too long to try to debug in the time period available to me[16].

I next had a look at the Taylor's Series method of calculating cosine and sine

and could have implemented this using multipliers, adders and dividers but apart from how inefficient this would be on an FPGA, it would also have taken a very long time to complete to a high enough standard.

In the end I settled for a look-up table type module that stored a value for each positive number between 0 and 1024 and returned the cosine and sine in floating point format. I instantiated this component in my Rotation module and then changed the sign of the sine result and registered the three outputs, cos, sin and msin (minus sine), to be utilized by my matrix builder component.

Display Stage

When I was happy with my first four components, I next had to implement the part of my project to handle the 3D to 2D projection of the transformed vectors to the screen. I decided it would be best to make a module that allowed the user to select the type of projection, from Orthographic or Perspective, with the ability to add other modes at a later date. For this I used a FOR GENERATE loop with IF GENERATE statements to call a port map of the Orthographic or Perspective component if that mode had been selected. This was necessary as components cannot be instantiated in normal FOR, IF or CASE statements.

The Orthographic or Parallel component was quite straight forward and basically registered the inputs and then outputted the values of X and Y and zero for Z and W.

For the Perspective component, I could have introduced one of the many perspective matrices available and passed the results through the multiplication and addition stage again but felt that this would add more complexity and delays to the design. Instead I opted to skip to the final stage of the simple perspective matrix, after the division across by homogenous W and work out the vector as shown below.

I made a Z Divider module that basically used the RARE project divider to return Z/d and d/Z (in case I wanted to revert back to the step before homogenous division). I used this in my perspective component and after registering my inputs, I instantiated two multipliers to calculate the first two coordinates of the final vector and outputted d and '1' for the second two coordinates. This then returned

$$\begin{bmatrix} \mathrm{d/Z}\times\mathrm{X} \\ \mathrm{d/Z}\times\mathrm{Y} \\ d \\ 1 \end{bmatrix}$$

Figure 3.8: Final Perspective Vector

the perspective projection results to the Display stage for outputting.

3.4.3 Conclusion

As you can see, my final design was much bigger than all others and forced me to research many different alternative approaches to solving my problems. Although this was frustrating at times, when I had to look through other peoples' code, trying to solve their problems and not succeeding, it was also a learning experience as it gave me insight into how other people approach hardware design. The whole exercise of trying to configure someone else's components to integrate with your own is as much a part of the design process as coming up with completely original ideas. As well as learning new coding techniques and elements of the VHDL language I had never seen before, it also felt like what it would be like to be part of a team of engineers, trying to compile our divided tasks together to produce a result, albeit on a much smaller scale.

Some of the aspects of the VHDL language I did not know before reaching this stage were things that I had never needed to use in the duration of my course, such as adding libraries through ModelSim, how to convert between number types and how to instantiate components depending on inputs using GENERATE statements. I also learned about the potential future of VHDL with such things as floating point packages becoming an IEEE standard.

I also learned a lot of the theory while doing this final design, including the many ways of calculating trigonometric functions like cosine and sine and how the most efficient way to implement this on an FPGA is the CORDIC algorithm.

I also read a lot of interesting information about the many different perspective matrices and the advantages and disadvantages of each method while trying to focus on which would be the most precise and practical for my design.

One crucial lesson I learned was about the importance of testing designs. Before this project I merely tested using a small selection of values and assumed correct functionality which is not enough for complex math functions such as vector transformations, as I discuss in the future work section. Before that though, I had one more design which attempted to compact my final design as the immense number of floating point multipliers, adders and dividers were too large to fit on the Spartan 3 XC 3S1000 board.

3.5 Compact Design

As I have said, my final design was too large for the board but I desperately wanted to see my design working in hardware as I thought it would give me a better sense of completion to see my design perform in hardware as well as in simulation. I decided to make a cut down version of my design that would obviously be less efficient than my full design but would fit on the board.

3.5.1 Design

As even one of the RARE multipliers, adders or dividers used up more than 100% of the Spartan 3 XC 3S1000's resources, my first task was to find a multiplier that would work efficiently on the project board. Since I was now aware of opencores, I found a floating point unit which was aimed at FPGAs which I integrated and tested successfully. Unfortunately, the author had not yet been successful with his divider module so I decided to cut my design back to the transformation stages. Another problem I still faced was that not all the multipliers and adders I required for my transformations would fit on the board as they still used up more than 100% of the board's resources collectively. This meant that I would have to use only one multiplier and one adder through which all products and sums must be performed. I therefore decided to change my multiplication and addition stages to new stages that would handle the multiplexing and de-multiplexing needed to do this.

3.5.2 Implementation

Multiplication Handler

My Mult Handler component basically took in all the same inputs as my previous multiplier stage but instead of instantiating sixteen multipliers, only one was instantiated.

The inputs of this multiplier were then connected to the output of a 20-to-2 multiplexer which selected a different set of inputs (which were X, Y, Z, W and all the elements of the matrix outputted from the Matrix Builder) depending on the value of the select line.

The outputs of the single multiplier component were then connected to a 2-to-19 de-multiplexer which outputted the correct result depending on the mode of its select line. Both multiplexer and de-multiplexer components' modes were configured so that in the top file I could connect a counter to the select lines to automate the process of passing values through the multiplier. This was made possible as this new multiplier module outputted it's result in the same clock cycle.

Addition Handler

My Add Handler was set up in a similar but slightly more complex fashion than my multiplication handler. As well as having a multiplexer and de-multiplexer either side of one lone adder component, some outputs of the de-multiplexer were put back into the multiplexer as shown in the schematic diagram. This was due to the temporary results returned from the adder in the addition stage that needed to be added again to get the full answer for each coordinate.

Matrix Builder and Rotation

In order to help my new compact design fit on the board I needed to remove the multipliers from the Matrix Builder component and use the Mult Handler to calculate the product of $Cos\alpha$ and Sx, Sy and Sz.

3.5.3 Conclusion

In short, I ran out of time before I could get this design working on the project board. This was a bit disappointing as I really would have liked to see if my design worked in hardware as it did in simulation. I suppose this is not really an issue since my final design synthesised perfectly, showing that it would work given a big enough FPGA and board to hold it.

I believe this design did not work because of circuit timing. I noticed that although the counters in the top file should have kept everything working together in sync, I must have missed one clock cycle somewhere in the design as it did not function correctly, although it did synthesise and would have fit on the project board. I tried to solve this problem by latching the outputs of the multiplication handler and only releasing all the results to the addition stage when all multiplications had been calculated. I then introduced another adder to the design (the FPGA still had enough space for a few more adders) to calculate the addition of the temporary results instead of trying to deal with the timing required to loop back into the same adder but unfortunately, before I could get this to work, I ran out of time.

I still enjoyed doing this design however, as it made me think of new ways to implement my final design, aimed at reduction of area rather than speed and efficiency, which is a big issue in hardware engineering. Although Xilinx provides tools to optimize code for speed or area, this wasn't enough as my previous design had been so huge. This meant I had to look at my design from different angles and decide upon the best way to make it as small as possible, which I believe I would have gotten working, time permitting.

This led me to learn an important lesson about the need to look ahead in a project and try to foresee problems that you are likely to encounter later, which I will be sure to remember in future projects. If I had identified the possibility of area problems earlier in my design process I could have taken steps to reduce these and might have succeeded in loading my design onto the project board.

Chapter 4

Evaluation

In this section I will discuss the testing of my final design and evaluate its ability, efficiency and accuracy.

4.1 Individual Components

Multiplication and Addition Stages

The majority of my testing time for the multiplication stage was spent on testing of downloaded modules. For each module I downloaded, I created a new project to test this individually and if errors were found I would attempt solve them either through my knowledge of VHDL or by searching the internet for other peoples' solutions when they had similar problems. All this was very time consuming and quite disappointing when you put time and energy into something and then cannot get it working and have to repeat this a couple of times in a row. But, as I have said, it is also a learning experience from which I have gained.

When I found the RARE project files I synthesised the modules and tested each with test bench waveforms with both positive and negative numbers. I didn't put too much emphasis on my test range here as I knew that plenty of test values would be put through both the adder and multiplier in the duration of the project and that inaccuracies would be easily spotted if faults occurred in the calculations.

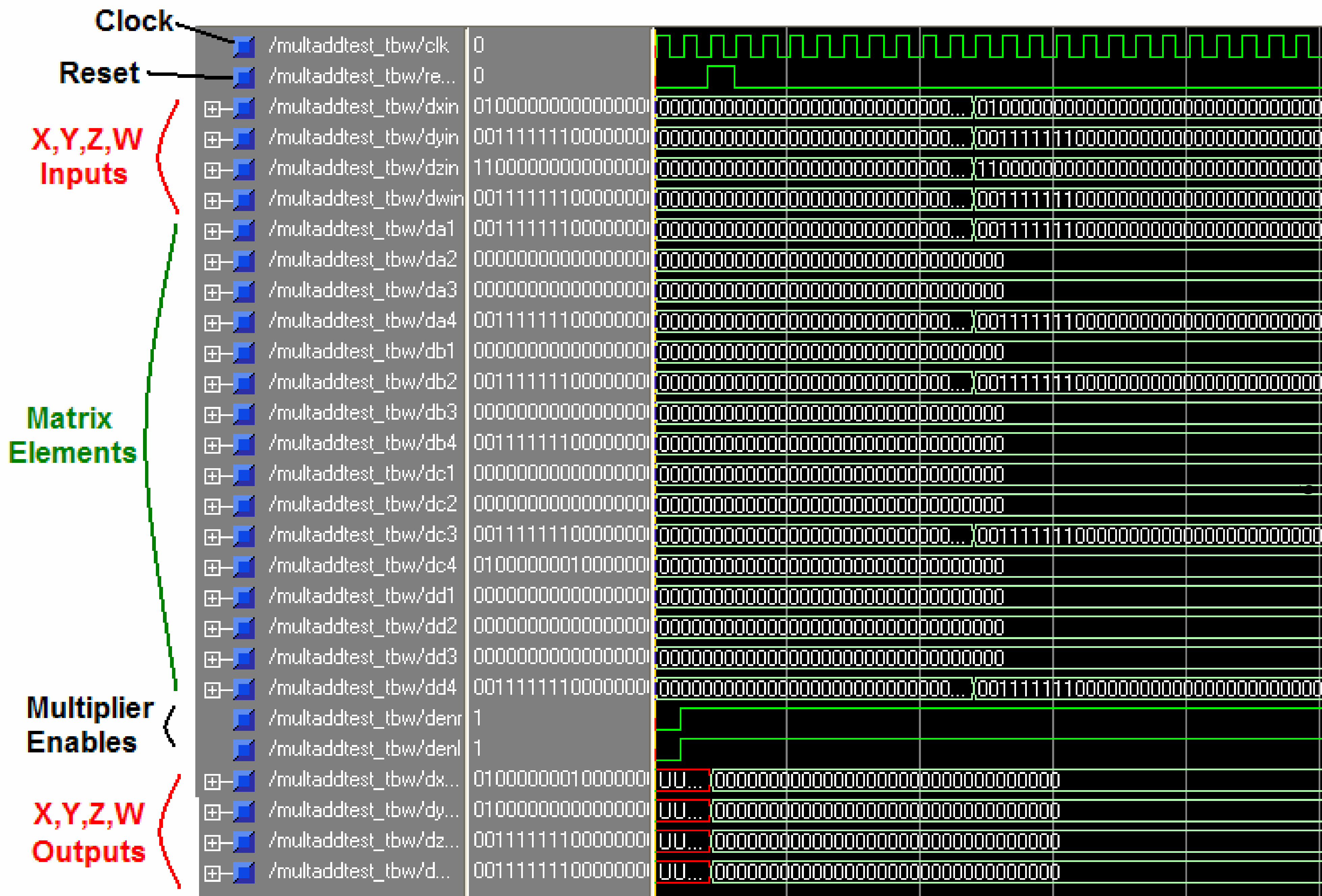
Before expanding both the multiplication and addition stages to floating point, I had to expand and retest my D Flip Flop which I was using to register the inputs

before they were sent to the multiplier/adder in each stage. I updated the test bench waveform to test the flip flop and then did the same for each of the two stages after I had instantiated the new multipliers and adders. Again I loaded these test benches with both positive and negative numbers to test. I discovered that both components worked fine, the Multiplication stage taking 8 clock cycles to produce a result and the Addition stage taking 17 clock cycles, making sense as the inputs are registered and then the adders are called twice.

I then made the Multaddtest component which instantiated both multiplication and addition stages and made a test bench to ensure that both worked correctly together, which they did after 26 clock cycles, as can be seen on the waveform on the next page.

On the left of this waveform you can see the input and output signals and the values loaded in them. As you can see the clock pulse is on the top, the reset line underneath that, the inputs under that and the outputs on the bottom. At point one you can see that the outputs are red until the reset goes high and low. This is because the values are Unknown until set to zero with the reset. At point one all input values are set to zero. At point two, you can see the 32 bit floating point values being loaded in for the inputs. The values for inputs X,Y,Z and W are 2.0, 1.0, -2.0 and 1.0 respectively. Inputs da1, db2, dc3 and dd4 are all set to 1.0. This corresponds to Sx,Sy,Sz and 1.0 for the homogenous row/column - practically making the identity matrix so as to not alter the input vector. However, Input da4, which corresponds to Tx in the transform matrix, is set to 1.0. This means that the calculation being carried out is a translate of one position on the X coordinate. As you can see, after 26 clock cycles, at point three, the outputs change to reflect the output from my two components. This is the vector (3.0, 1.0, -2.0, 1.0) which is exactly what was expected.

This was just a small test and some more values were tried with these components during my evaluation.



Clock
Reset
X,Y,Z,W Inputs
Matrix Elements
Multiplier Enables
X,Y,Z,W Outputs
/multaddtest_tbw/clk
/multaddtest_tbw/re...
/multaddtest_tbw/dxin
/multaddtest_tbw/dyin
/multaddtest_tbw/dzin
/multaddtest_tbw/dwin
/multaddtest_tbw/da1
/multaddtest_tbw/da2
/multaddtest_tbw/da3
/multaddtest_tbw/da4
/multaddtest_tbw/db1
/multaddtest_tbw/db2
/multaddtest_tbw/db3
/multaddtest_tbw/db4
/multaddtest_tbw/dc1
/multaddtest_tbw/dc2
/multaddtest_tbw/dc3
/multaddtest_tbw/dc4
/multaddtest_tbw/dd1
/multaddtest_tbw/dd2
/multaddtest_tbw/dd3
/multaddtest_tbw/dd4
/multaddtest_tbw/denr
/multaddtest_tbw/denl
/multaddtest_tbw/dx...
/multaddtest_tbw/dy...
/multaddtest_tbw/dz...
/multaddtest_tbw/d...
1
2

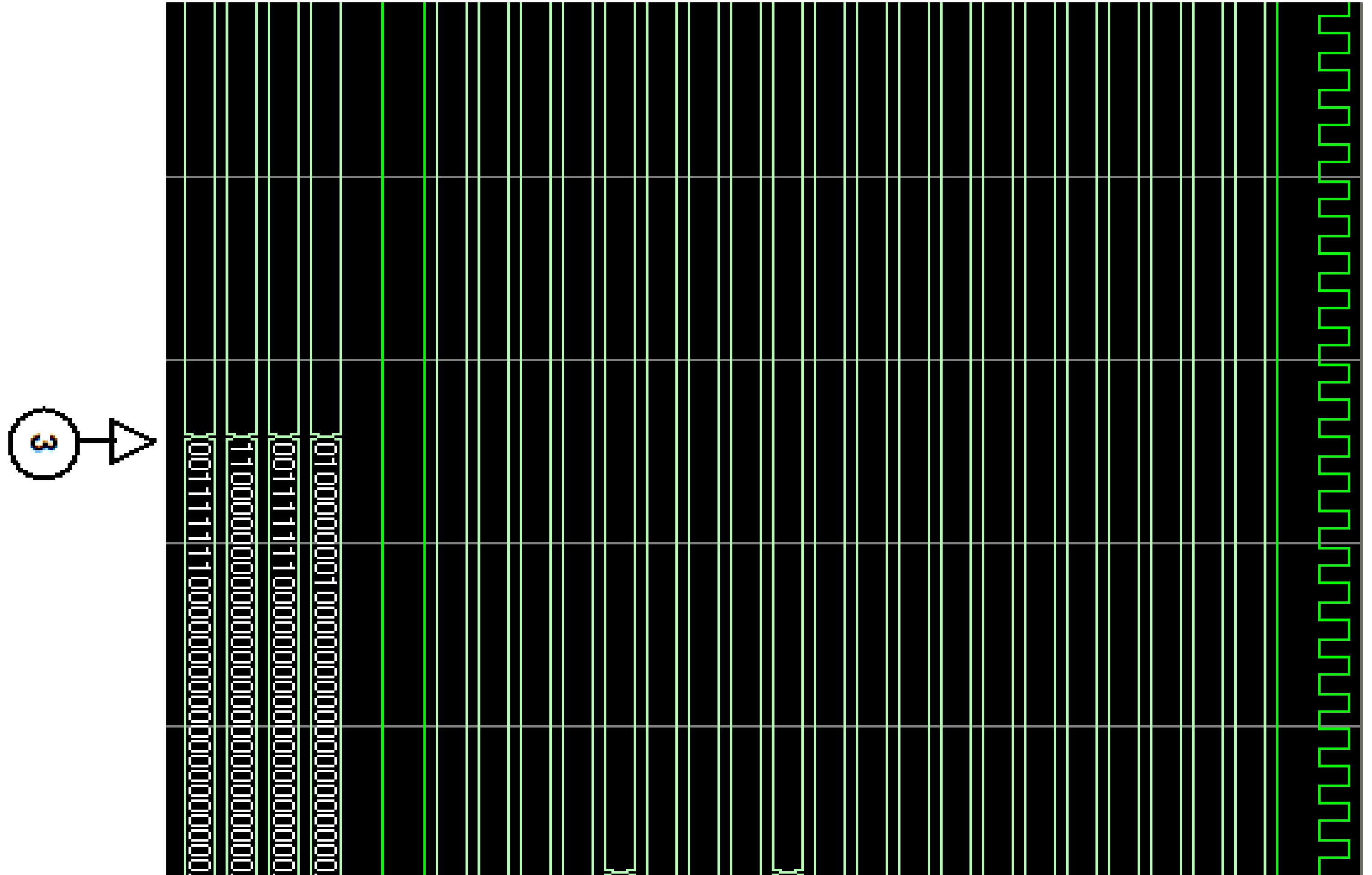


Figure 4.2: Multaddtest Test Bench Waveform - Part 2

Matrix Builder and Rotation Component

After making my Matrix Builder component I made a test bench waveform to test the different RVST modes. When I ran it, the first RVST mode worked fine but when I switched mode during the waveform, the output matrix never changed. I discovered that this was due to me using a case statement and the module getting stuck in one mode. I converted this case statement into a series of IF statements and retested. This solved the problem as each IF statement was tested individually, testing for each mode, rather than the case statement testing RVST once and staying in that state. I confirmed that all of the modes were now working using a whole number for $Cos\alpha$ so that I could see that the multiplication in my matrix builder module was correct and loading the result into the proper matrix position when required.

I then made the Testtoadd component which brought together the Matrix Builder, Multiplication and Addition Stages for testing in one waveform. I inputted some test values and now had my design loading, multiplying and adding matrices correctly.

The waveform on the next page shows one such calculation of test values. As you can see on the left the inputs are labelled again, going from clock to reset, through the inputs and the outputs are on the bottom.

At point one, there is a reset and values are loaded into the inputs on the second cycle (as reset prevents the values from loading). As you can see the input vector is (1.0,1.0,1.0,1.0). The RVST mode is set to 0011, meaning there is a scale and translation taking place. Tx is set to 3.0 and the scales are set to 1.0 except Sy which is 2.0. As you can see at point two the outputs reflect the expected answer of (4.0,2.0,1.0,1.0).

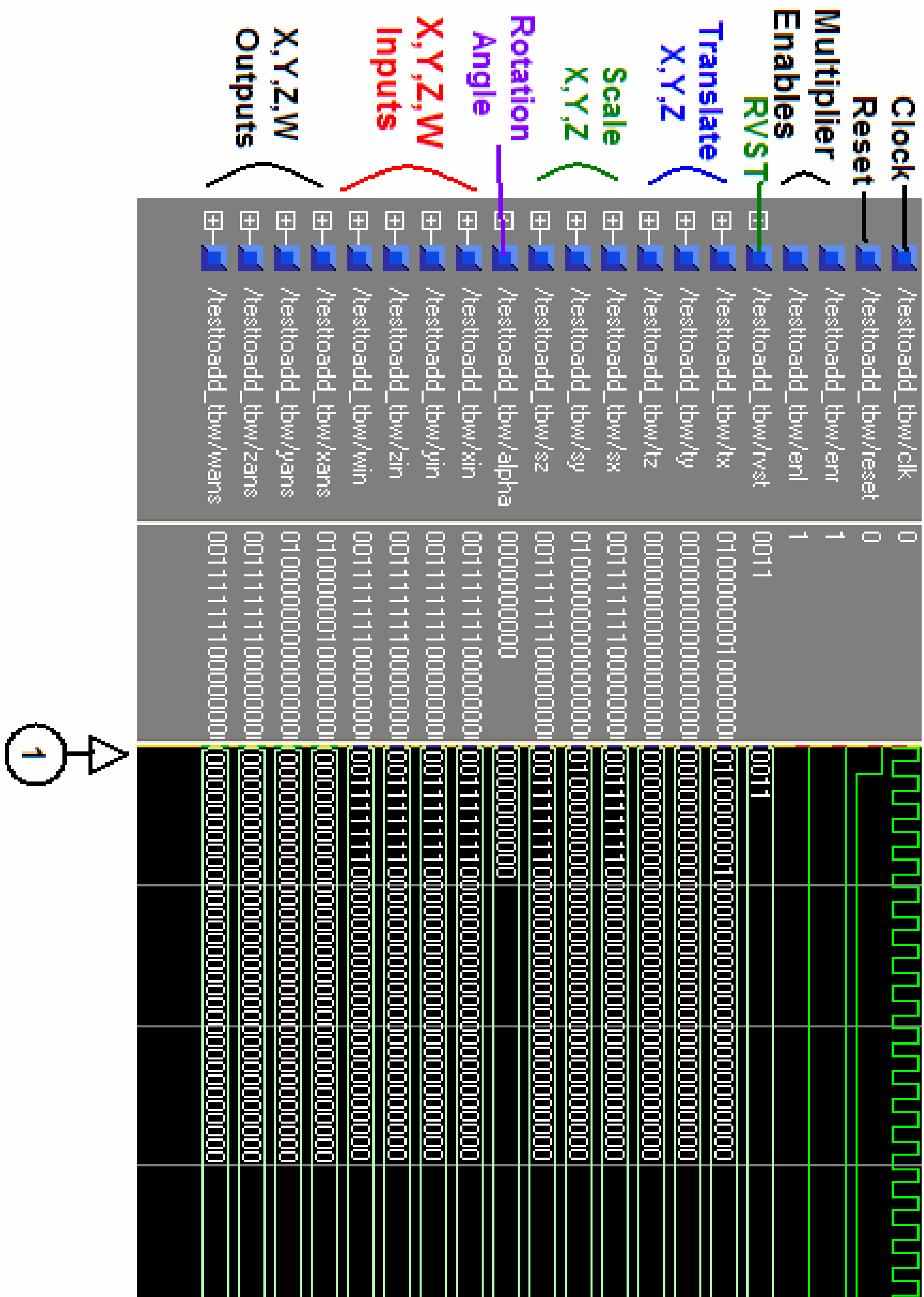


Figure 4.3: Testtoadd Test Bench Waveform - Part 1

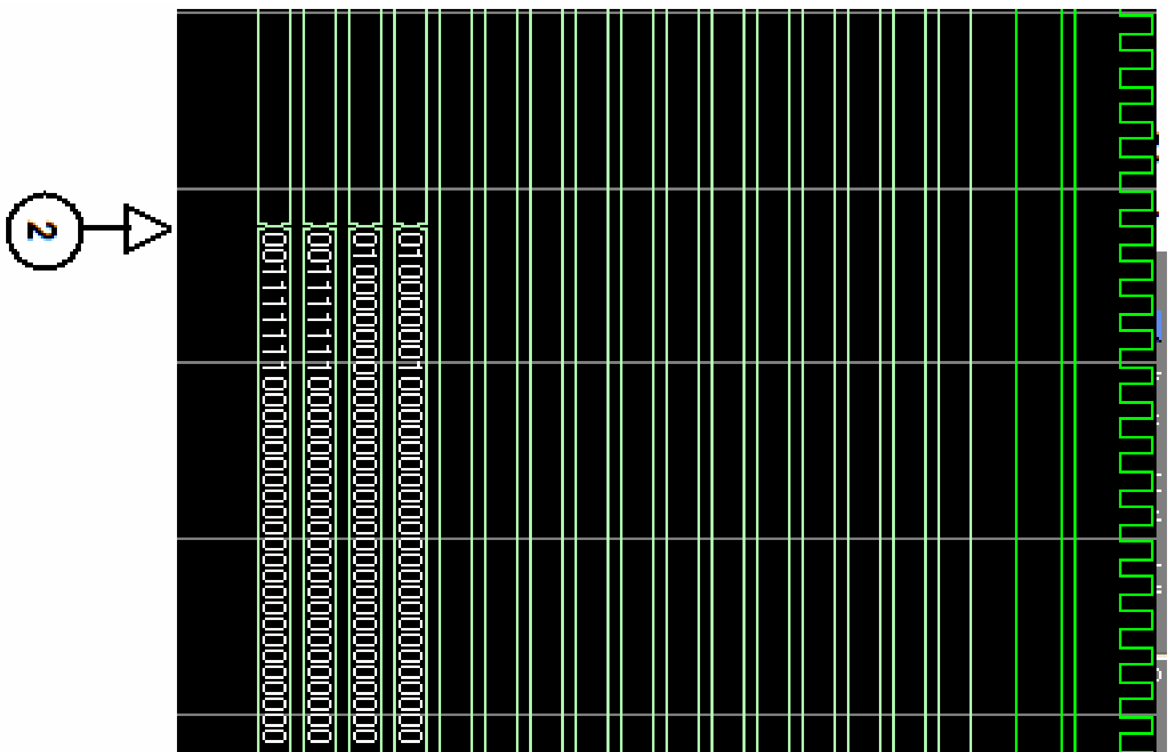


Figure 4.4: Testtoadd Test Bench Waveform - Part 2

For my Rotation component, most of the testing time was spent trying to get a module to calculate trigonometric functions working properly. So much so that when I finally settled for the look-up table approach which came with the other RARE project files, it was very late in my design process and came as a surprise to me when I realized that the look up table was returning incorrect answers. I think this was because the module was aimed at calculating cosine and sine for Discrete Fourier Transforms and expected a different form of input instead of the angle, which was required of my design. In the end I modified this look-up table to hold cosine and sine values in floating point of all angles from 0 to 360 in steps of 15, as these were the most common rotations. This obviously did not fulfil the necessary level of rotation required for 3D graphics, but sufficed for testing of my integrated components. Time permitting, I could have stored values for angles in increments of one but this is just trivial work as the actual calculation of trigonometric functions was not a priority in my design and any working module can easily be inserted in place of my ”mycossin” module to perform correct calculations.

After making this mycossin component, I made a test bench waveform and ensured that is was functioning correctly.

Display Stage

Before testing my display stage, I first tested each subcomponent. The waveform for the parallel/orthographic stage was quite straight forward and worked without delay. The Z divider module took seventeen clock cycles to return a correct result as division is much more computationally expensive than multiplication. As the perspective stage involved both this division and a multiplication (8 clock cycles), the test bench waveform for the Simple Perspective module took 26 clock cycles to return the correct result, the extra clock cycle being the registering of the inputs before being mapped to instantiated components.

I then tested the Display Mode component to make sure it changed with the mode select and found that everything functioned correctly.

4.2 Integrated Components

Now that I had each component tested and confirmed working for some test values individually, it was time to test my design altogether. I made a top file that instantiated each of my four main stages and my Rotation component and a test bench waveform for this top file. I first tried a simple scale and translate of the input vector (RVST = 0011) and was surprised to see the wrong answers come out as each stage had worked perfectly individually. I stepped through each component in my head and realised that my problem was with the two vectors I hadn't scaled being reduced to zero. I then realised my mistake of leaving the unused scale input values at zero. This had the obvious effect of multiplying the input vector coordinates not being scaled, X and Z by zero, making them zero for the rest of the components and leading to the wrong answers. I then set the scale value inputs that I didn't want to scale to 1.0 and my design returned the correct result. I tried some more input values for this RVST mode, ensuring that each answer was correct by verifying the result with Octave.

One example of this RVST mode is shown in the waveform on he next page. The input vector is (2.0,2.0,2.0,1.0). Tx is set to 4.0, Sy is set to 2.0, and the other scales are left at 1.0. The Display mode is set to 01, which is perspective projection and the distance to the front clipping plane, d is set to 1.0. As there are so many sections taking 20+ clock cycles, the results take some time to appear. In the waveform I have skipped ahead to the results. At point two we can see that output coordinate Z is set to the distance, d, which was 1.0. At point three some temporary results begin to be outputted by the divider/multiplier but we need to wait the full 26 clock cycles needed by the final division and multiplication, until point four, to see the final result. This is (3.0,2.0,1.0,1.0) which is correct as X had been translated up to 6.0, Y had been scaled to 4.0, but both were then multiplied by 1.0/2.0 (d/Z), giving the correct results.

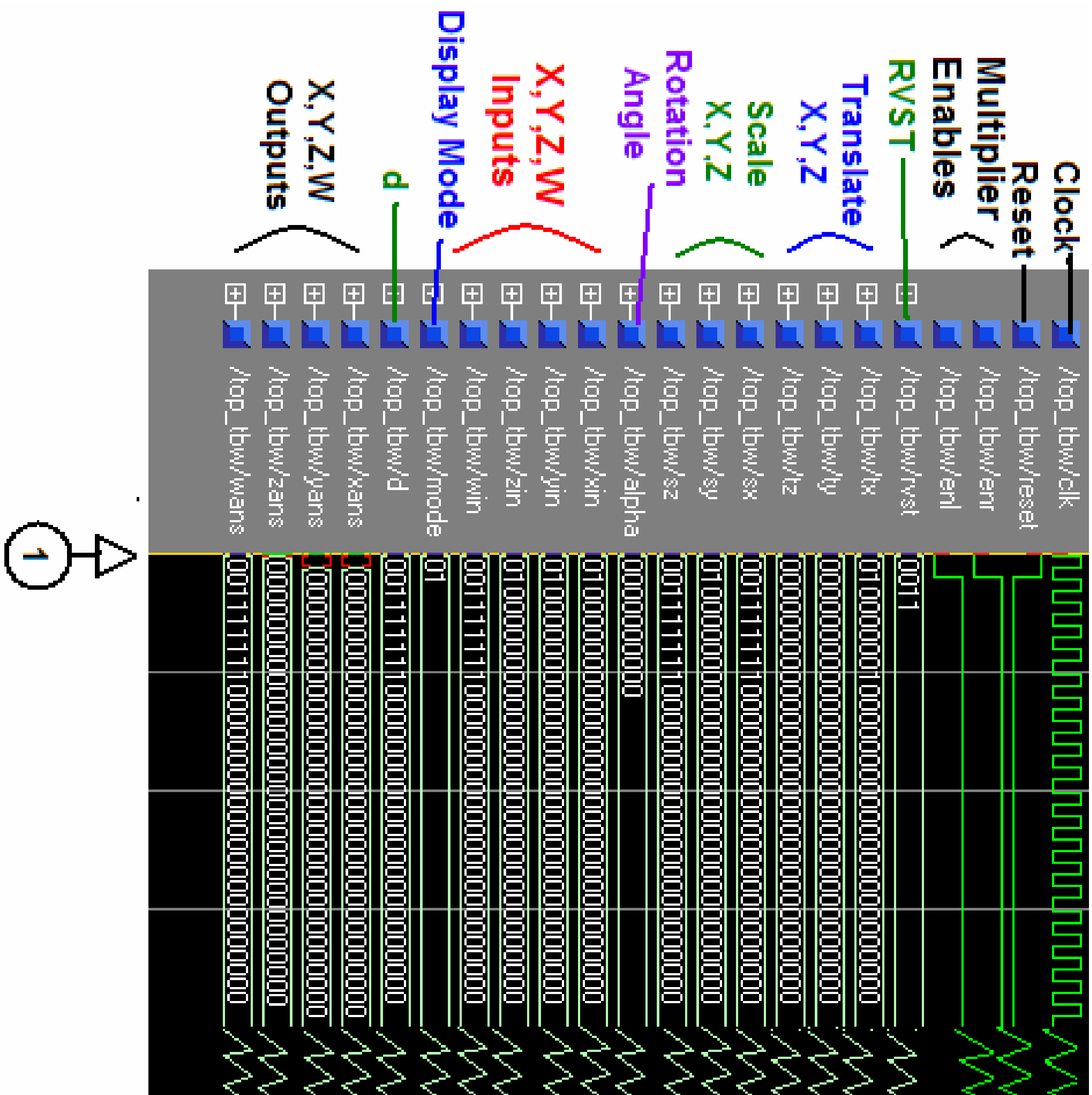


Figure 4.5: Final Test Bench Waveform - RVST = 0011 - Part 1

Figure 4.6: Final Test Bench Waveform - RVST = 0011 - Part 2

Next I tried a rotation of the Z axis (RVST = 1100), rotating the input vector (2.0,2.0,2.0,1.0) 180 degrees in an anti-clockwise direction. As you can see the input alpha is set to the binary of 180. Translates are set to 0.0, scales set to 1.0 and the display mode as in the previous example. At point two the mode is switched to Rotation of the Z axis. This time delay between setting the inputs and selecting the mode is to give the multiplier in the Matrix Builder stage time to multiply the Scale inputs by the Cosine of Alpha. Skipping ahead again in the testbench we can see the correct results come out as (-1.0,-1.0,1.0,1.0) This is correct as if you imagine the point (2,2,2) it is in the first quadrant, where X and Y is positive. Rotating this 180 degrees either clockwise or anti-clockwise around the Z axis leaves the point in the third quadrant, where X and Y are negative. The reduction of the X and Y outputs are again due to multiplication by $1.0/2.0$.

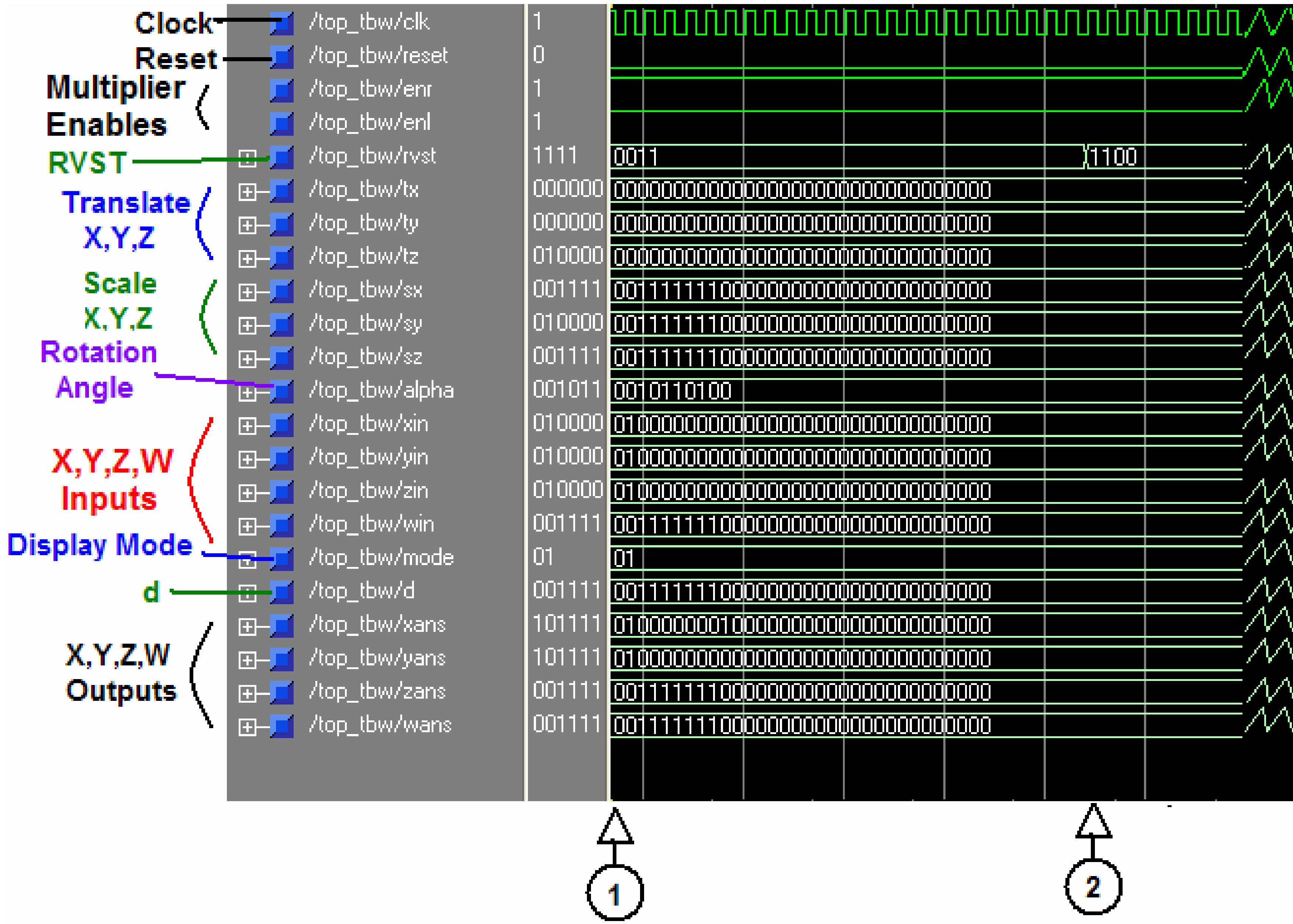


Figure 4.7: Final Test Bench Waveform - RVST = 1100 - Part 1

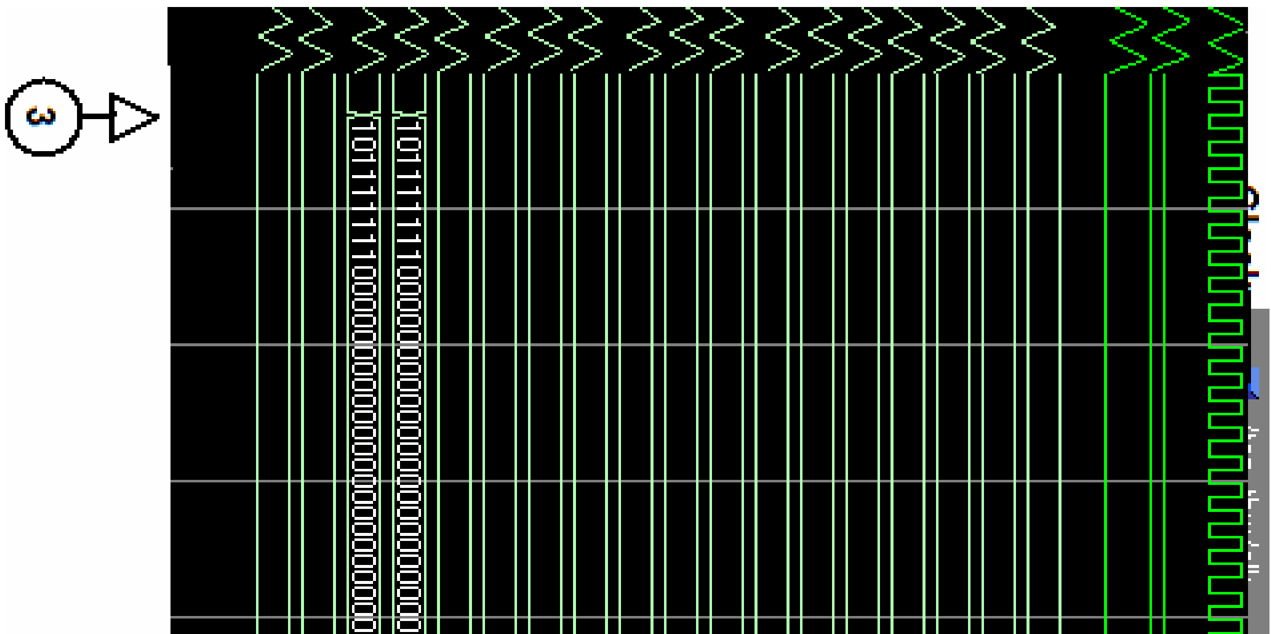


Figure 4.8: Final Test Bench Waveform - RVST = 1100 - Part 2

I kept the same input vector and rotation axis and degree but added scale and translate to test all transforms at once (RVST = 1111). I set Tz to 2.0 and Sy to 2.0, leaving the rest as their default values. Again I delay the mode change until point two and you can see temporary results at point three. The final results at point four are (-0.5,-1.0,1.0,1.0). This is again the correct result as Y had been scaled to 4.0 and Z had be translated to 4.0, making the fraction to be multiplied in the perspective component now $1.0/4.0$. As you can see the output coordinate X is one quarter its original value, Y is one quarter its scaled value and Z is still set to the value of d. The signs of the output vector are also changed due to the rotation of 180 degrees around the Z axis as explained in the last example.

Although these are nearly all the tests I got the chance to perform, I believe that they show my design to be working perfectly for the small set of test values, although not optimized. As well as Octave, I also used a handy java applet[17] to test these results.

Figure 4.9: Final Test Bench Waveform - RVST = 1111 - Part 1

Figure 4.10: Final Test Bench Waveform - RVST = 1111 - Part 2

Chapter 5

Future Work and Conclusions

In this section I discuss how I would complete my project, expand on it and the benefits of my project for others approaching a similar design. I also conclude my project and talk about the experiences I have gained in its duration.

5.1 Completion

Although I achieved a lot in the course of my project, it is clear that my design is not perfect. Although my project worked perfectly for some test values, to call it complete I must first and foremost assure its functionality by testing it with a much wider range of input values

Ideally, if I had more time, I would have created a testbench file that would load a series of different input values into my design and test them against a set of correct answers like how some other students verified their projects. This would have helped me to discover any bugs that my simple tests did not show and would need to be ironed out before my design could be considered complete.

One bug in my design I did notice was similar to the scale problem I overlooked earlier. When scale was set to zero it changed the input coordinate to zero for the rest of the calculation and led to incorrect results. Although I noticed this and remedied it by setting the unused scale to 1.0, I overlooked the effect of rotating by certain angles. The problem here was that the cosine or sine of some angles returned the value zero. This was obviously necessary for some rotations

to be accurate but if they occurred at the same time a scale was to occur the scale would be multiplied by zero and lost. This was obviously why no books or websites had combined the three transform matrices into one, although none of the sources I read explained why. This problem could obviously be remedied with the separation of the matrix builder component for handling scales and translates OR rotations and translates. This would solve the problem but lead to a much slower design if one wished to perform all three transforms at once. A better solution, which would keep all the transforms combined and hence the efficiency, would be to add another small component with a zero detector between the Rotation and Matrix Builder components. This would also be connected to the scale inputs and could determine from these if a collision of a scale value and a zero cosine or sine value would occur. It could then take steps to avert an incorrect result, either by passing the zero cosine/sine value through as 1.0 or, if this affected the rotation, separating the rotate and scale into two separate calculations, re-combining the results after the Addition stage and continuing to the Display stage. Although this second solution would add extra time for some particular cases that caused a collision, it would remain efficient for all other cases and have no incorrect results.

Another area of my design that obviously needs improvement is the Rotation component or specifically the trigonometric function calculator. For this I would attempt to obtain a working module of the CORDIC algorithm or even try to code one myself.

While improving components I would also try to implement a more efficient floating point unit as although the seven clock cycles taken by the RARE project adder and multiplier were efficient enough for this project, I am sure that there exists a faster, more accurate FPU, especially with a more efficient division unit.

Although in my design I separated each component and registered the inputs to each stage to add parallelism, I never tested the efficiency of this, again due to time restrictions, and assume that a lot more calculations could be preformed per second if more parallelism was introduced.

As you can see there are a number of areas in which my design could be improved, especially to ensure perfect functionality and then to make it more efficient. It would be interesting to see if I could complete my design to such a level that it could be utilised in the Trinity College graphics card research but even if I

do not decide to complete my project, the door is always open for other people to continue on my progress.

As well as paving a way for others to take over, my project also benefits anyone undertaking a similar project. If someone decides to create a model to perform graphics transformations and projections on an FPGA, or similar programmable graphics project, I feel this report would be very beneficial to them. Regardless of their design, if such a person read this report he would be aware of problems that I faced and can take action to prevent the same problems occurring in the course of their design. In this way other people can basically learn from my mistakes, which is better than having to make them yourself first.

Also, since I attempted several different designs that ultimately were not useful to my final design, this would prevent people from wasting time and energy exploring these roots unless, of course, they had ideas to overcome the problems I faced.

5.2 Expansion

If I did perfect my design the opportunities for expansion are endless. I could examine other forms of projection such as axonometric projections or perspective projections with more than one viewpoint. I could even implement the more difficult perspective projection matrices that used trigonometric functions such as Cot and Tan to perform normalization and clipping as well as more complex perspective.

It would also be interesting to implement a whole mini graphics pipeline working together, if only in simulation. This could first translate the camera, perform normalization and clipping and set up the scene before putting the objects in and drawing the scene.

5.3 Experiences

Overall, I found this project very interesting even though I had anticipated not liking the mathematical aspects of 3D graphics. I am very glad I took Mr. Manzke's

advice and attempted this project for if I had not accepted this project I may have ended up doing something of no interest to me. Looking at graphics in the way I did for this project totally changed my outlook, from disliking them in high level coding such as C++ and Open GL, to actually wanting to complete a project about them in my spare time. This project has not only helped me gain a lot of knowledge on 3D graphics and the maths behind them, but also experience in hardware design which has confirmed my consideration as to my future career and the direction I wish it to take. I can certainly see myself being involved with hardware in my career, maybe mixed with my other interest, mobile communications. The actual hardware programming experience I have gained during this project has improved my VHDL coding standard to a stage I never thought possible in such a short period of time and I now have a much greater confidence in my ability, not only with regards to VHDL, but to undertaking projects in general. I have also learned some indispensable lessons both in hardware and in general, about the importance of research, forward planning, documentation and scheduling which I will carry with me forever. Another vital aspect of projects that I learned is that although you can complete 95% of a project with relative ease, the last 5% is amazingly difficult to complete, especially if working with a deadline. This is why scheduling is so important and I will be sure to put more emphasis on the time needed to complete the final stages of a project in future.

Apart from not getting to fully perfect my project, my only other disappointment was not getting to see my design working in hardware as, personally, I always fell more satisfied when I have a finished product to look at. I think this is one of the reasons hardware engineering appeals to me, as your final product can be seen performing a task in an actual device, rather than being just software on a computer screen. Hopefully I can experience this satisfaction with this project at a future date. That aside, I can honestly say I enjoyed this project and the experiences I have gained from it. I would once again like to thank Michael Manzke for giving me the chance to undertake a hardware project and for all the help over the last seven months.

5.4 Conclusion

As can be seen from the evaluation of my design, FPGAs are definately capable of complex calculations such as 3D vector transformations and projections. This shows that they could be used to assist GPU and CPU setups for Graphics and possibly other applications. Although my project proves this, the important aspect of whether this will add efficiency to the total design has yet to be seen. The key issue to decide this is the efficiency of the individual components of my design. I truly believe that with the proper floating point unit and trigonometric calculator my design could work alongside the GPU and provide some relief for some calculations, maybe not performing calculations as fast as the GPU, but still achieving results. While the FPGA is working on these values, the freed GPU timeslots could be used for other calculations, thus speeding up the design as a whole.

Although this seems the case, the only way to test this hypothesis is through further research and testing of this design (obviously after perfect functionality is achieved) in the proper graphics environment such as by the Trinity College graphic research group. This task could even be a future computer science post graduate research project, which I would definitely like to see happen, even if I could not be a part of it.

5.5 Attached CD

Attached to this report is a CD containing my source code and a PDF copy of this report. My final design is commented and seperated into three main parts: multaddtest, testtoadd and top - which contains everything. These files take quite some time to synthesize as they are so large and the test bench waveforms require a few minutes to reach the end.

Bibliography

[1] Xilinx. *Using Block RAM in Spartan-3 Generation FPGAs*. http://www.xilinx.com/bvdocs/appnotes/xapp463.pdf, March 2005.

[2] *3D Axes - X,Y,Z Plane - axis.gif*. http://www.wonko.info/vrml/vrml1.htm.

[3] *3D Cows*. http://www.dcs.shef.ac.uk/graphics/research/multires/.

[4] *3D Rendering Pipeline: The Geometry Stage - pipeline.gif*. http://www.willamette.edu/g̃orr/classes/GeneralGraphics/Pipeline/geometry.htm.

[5] *3D Viewing Frustum - frustum1.gif*. http://astronomy.swin.edu.au/ pbourke/projection/frustum/.

[6] *One, Two and Three Point Perspective*. http://www.atpm.com/9.09/design.shtml.

[7] Steven. Winikoff. *Viewing in 3D*. http://www.cs.concordia.ca/ comp471/slides/Viewing_3D.pdf, January 2005.

[8] Latha. Pillai. *Matrix Math, Graphics, and Video*. http://direct.xilinx.com/bvdocs/appnotes/xapp284.pdf, October 2001.

[9] *Orthographic projection images*. http://en.wikipedia.org/wiki/Orthographic.

[10] Dr. Gerda. Kamberova. *Math of Projections*. http://cs.hofstra.edu/courses/2003/fall/csc171/A/handouts/, 2003.

[11] Ann. McNamara. *Viewing systems & OpenGL*. https://www.cs.tcd.ie/courses/baict/bass/4ict10/Michealmas2002/Handouts/17_Viewing&OpenGL December 2002.

[12] Timothy. Miller. *The Open Graphics Project*. http://lists.duskglow.com/mailman/listinfo/open-graphics, April 2005.

[13] Charles R. Mano, Morris M. & Kime. *Logic and Computer Design Fundamentals Second Edition Updated*. Prentice-Hall, Inc., 2001.

[14] Howard University Electrical and Computer Engineering Department. *A Remote Adaptive-Computing REsource on the Internet(RARE)*. http://angelou.imappl.org/ cgloster/rare/.

[15] D. Bishop. *Floating-Point HDL Packages*. http://www.eda.org/fphdl/.

[16] Richard. Herveille. *CORDIC core*. http://www.opencores.org/cores/cordic/, September 2001.

[17] Alain. Guyot. *Floating-point addition*. http://tima-cmp.imag.fr/ guyot/-Cours/Oparithm/english/Flottan.htm.

[18] Steve M. Slaby. *Fundamentals of Three-dimensional Descriptive Geometry Second Edition*. Wiley, 1976.

[19] Lars. Kadison. *Projective geometry and modern algebra*. Birkhuser, 1996.

[20] C. Ray (Clarence Raymond). Wylie. *Introduction to projective geometry*. McGraw-Hill, 1970.

[21] E. A. (Edwin Arthur). Maxwell. *Geometry by transformations*. Cambridge University Press, 1975.

[22] James T. Smith. *Methods of geometry*. Wiley, 1999.

[23] Michael E. Mortenson. *Geometric Modeling*. Wiley Computer Publishing, 1997.

[24] Howard. Anton. *Elementary Linear Algebra*. John Wiley, 2000.

[25] Thomas. Mlhave. *Transformations in 3D*. http://home10.inet.tele.dk/moelhave/tutors/3d/transformations/transformations.html, October 2000.

[26] Diana. Gruber. *The Mathematics of the 3D Rotation Matrix*. http://www.makegames.com/3drotation/, September 2000.

[27] Prof. John. Robinson. *VHDL Quick Help Guide*. http://www.icaen.uiowa.edu/ vlsi1/web_131_dir/vhdl_help.html, August 2002.

[28] Weijun Zhang. *VHDL Tutorial: Learn by Example*. http://csold.cs.ucr.edu/content/esd/labs/tutorial/, June 2001.

[29] Department of Computer Science & Electrical Engineering UMBC. *VHDL reference material*. http://www.csee.umbc.edu/help/VHDL/, February 2004.

[30] Jan. Van der Spiegel. *VHDL Tutorial*. http://www.seas.upenn.edu/ ee201/vhdl/vhdl_primer.html#_Toc526061353, September 2001.

[31] Norbert. Lindlbauer. *The CORDIC-Algorithm for Computing a Sine*. http://www.cnmat.berkeley.edu/ norbert/cordic/node4.html, January 2000.

[32] Ray. Andraka. *A survey of CORDIC algorithms for FPGAs*. http://www.andraka.com/cordic.htm, February 1998.

[33] Rajendar. Yalamanchi, Mani Sudha. & Koltur. *Single Precision Floating Point Unit*. http://www.ee.pdx.edu/ mperkows/-CLASS_VHDL/VHDL_CLASS_2001/Floating/projreport.html, July 2001.

[34] G Scott. Owen. *Perspective Viewing Projection*. http://www.siggraph.org/education/materials/HyperGraph/viewing/view3d/perspect.htm, February 2005.

[35] Answers.com. *3D projection*. http://www.answers.com/topic/3d-projection.

[36] Thomas. Mlhave. *The Viewing Process*. http://home10.inet.tele.dk/moelhave/tutors/3d/viewing/viewing.html, Novemeber 2000.

[37] Emmanuel. Agu. *Projection*. http://www.cs.wpi.edu/ emmanuel/courses/cs4731/slides/lecture14.pdf, 2004.

[38] Wikipedia. *Graphics processing unit*. http://en.wikipedia.org/wiki/Graphics_processing_unit, April 2005.

[39] John W. Eaton. *GNU Octave*. http://www.octave.org/, 1998.

[40] Genevieve B. Orr. *3D Rendering Pipeline: The Geometry Stage*. http://www.willamette.edu/ gorr/classes/GeneralGraphics/Pipeline/geometry.htm, 2004.

[41] Pavel. Zemcik. *Hardware Acceleration of Graphics and Imaging Algorithms Using FPGAs*. http://www.fit.vutbr.cz/ zemcik/publics/sccg2002.pdf, 2002.