

REAL TIME 3-D GRAPHICS PROCESSING HARDWARE DESIGN USING FIELD-PROGRAMMABLE GATE ARRAYS.

by

James Ryan Warner

B. S. in Computer Engineering, Pennsylvania State University, 1999

Submitted to the Graduate Faculty of
Swanson School of Engineering in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH
SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

James Ryan Warner

It was defended on

September 18, 2008

and approved by

Dr. Alexander Jones, Assistant Professor, Department of Electrical and Computer
Engineering

Dr. Allen Cheng, Assistant Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Dr. James T. Cain,

Professor Emeritus, Department of Electrical and Computer Engineering

Copyright © by James Ryan Warner

2008

REAL TIME 3-D GRAPHICS PROCESSING HARDWARE DESIGN USING FIELD-PROGRAMMABLE GATE ARRAYS

James Ryan Warner, M.S.

University of Pittsburgh, 2008

Three dimensional graphics processing requires many complex algebraic and matrix based operations to be performed in real-time. In early stages of graphics processing, such tasks were delegated to a Central Processing Unit (CPU). Over time as more complex graphics rendering was demanded, CPU solutions became inadequate. To meet this demand, custom hardware solutions that take advantage of pipelining and massive parallelism become more preferable to CPU software based solutions. This fact has lead to the many custom hardware solutions that are available today.

Since real time graphics processing requires extreme high performance, hardware solutions using Application Specific Integrated Circuits (ASICs) are the standard within the industry. While ASICs are a more than adequate solution for implementing high performance custom hardware, the design, implementation and testing of ASIC based designs are becoming cost prohibitive due to the massive up front verification effort needed as well as the cost of fixing design defects.

Field Programmable Gate Arrays (FPGAs) provide an alternative to the ASIC design flow. More importantly, in recent years FPGA technology have begun to improve in performance to the point where ASIC and FPGA performance has become comparable. In addition, FPGAs address many of the issues of the ASIC design flow. The ability to reconfigure FPGAs reduces the upfront verification effort and allows design defects to be fixed easily.

This thesis demonstrates that a 3-D graphics processor implementation on an FPGA is feasible by implementing both a two dimensional and three dimensional graphics processor prototype. By using a Xilinx Virtex 5 ML506 FPGA development kit a fully functional wireframe graphics rendering engine is implemented using VHDL and Xilinx's development tools. A VHDL testbench was designed to verify that the graphics engine works functionally. This is followed by synthesizing the design on real hardware and developing test applications to verify functionality and performance of the design. This thesis provides the ground work for pushing forward the use of FPGA technology in graphics processing applications.

TABLE OF CONTENTS

| | |
|---|------------|
| PREFACE..... | XIX |
| 1.0 INTRODUCTION..... | 1 |
| 1.1 OVERVIEW..... | 1 |
| 1.2 STATEMENT OF THE PROBLEM..... | 7 |
| 1.3 OUTLINE..... | 9 |
| 2.0 COMPUTER GRAHPICS RENDERING..... | 10 |
| 2.1 MATHMATICS OVERVIEW | 12 |
| 2.1.1 Homogenous vectors..... | 12 |
| 2.1.2 Coordinate System..... | 13 |
| 2.1.3 Object Representation..... | 15 |
| 2.1.4 Affine Geometric Transformations..... | 16 |
| 2.1.4.1 Translation Transformation | 17 |
| 2.1.4.2 Scaling Transformation | 18 |
| 2.1.4.3 Rotation Transformation | 20 |
| 2.1.4.4 Transformation Compositions | 22 |
| 2.2 THE GRAPHICS PIPELINE..... | 26 |
| 2.2.1 Object Definition..... | 27 |
| 2.2.2 Scene Composition using World Coordinate Transformation..... | 28 |

| | | |
|---------|--|-----|
| 2.2.3 | View Coordinates and the View Transformation..... | 29 |
| 2.2.4 | 3D Projections and the Clipping Transformation..... | 34 |
| 2.2.4.1 | Perspective Projection | 37 |
| 2.2.4.2 | Parallel Projection..... | 39 |
| 2.2.5 | Clipping | 41 |
| 2.2.5.1 | Cohen-Sutherland Two Dimensional Clipping | 41 |
| 2.2.5.2 | Cohen-Sutherland Three Dimensional Clipping..... | 45 |
| 2.2.6 | Screen Coordinate Transformation..... | 49 |
| 2.2.7 | Rasterization | 51 |
| 3.0 | THE GRAPHICS PROCESSING UNIT | 56 |
| 3.1 | GRAPHICS PIPELINE | 61 |
| 3.1.1 | Matrix Multiplier Accelerator..... | 64 |
| 3.1.2 | Clipping Design..... | 67 |
| 3.1.3 | Line Rasterization..... | 79 |
| 3.1.4 | Frame Buffer and Display Interface..... | 83 |
| 3.2 | CENTRAL PROCESSING UNIT..... | 87 |
| 3.2.1 | Graphics Pipeline Control Registers..... | 89 |
| 3.2.2 | Other Peripherals | 90 |
| 4.0 | GRAPHIC PROCESSING UNIT IMPLEMENTATION AND TESTING | 91 |
| 4.1 | HARDWARE DEVELOPMENT PLATFORM..... | 91 |
| 4.2 | GRAPHICS PROCESSING UNIT IMPLEMENTATION..... | 94 |
| 4.2.1 | Floating Point Primitives | 96 |
| 4.2.2 | Microblaze Implementation..... | 100 |

| | | |
|---------|--|-----|
| 4.2.2.1 | Base System Builder..... | 106 |
| 4.2.2.2 | DVI IIC PLB Interface..... | 116 |
| 4.2.2.3 | N64 PLB controller interface..... | 117 |
| 4.2.2.4 | Graphics Pipeline Registers PLB interface..... | 119 |
| 4.2.3 | Graphics Pipeline Implementation | 120 |
| 4.2.3.1 | Floating Point Conversion and Matrix Selector..... | 121 |
| 4.2.3.2 | Matrix Transformation and Selection..... | 122 |
| 4.2.3.3 | Cohen-Sutherland Clipping..... | 123 |
| 4.2.3.4 | Bresenham’s Line Rasterizer..... | 124 |
| 4.2.3.5 | Frame Buffer and the ZBT Memory Controller..... | 125 |
| 4.2.3.6 | VGA Display Interface and the Line Doubler..... | 125 |
| 4.3 | GRAPHICS PIPELINE FUNCTIONAL TESTBENCH..... | 128 |
| 4.4 | GPU SYNTHESIS | 134 |
| 4.4.1 | Xilinx EDK and Microblaze | 134 |
| 4.4.2 | ISE and Full GPU Synthesis..... | 136 |
| 4.4.3 | Synthesis Results..... | 137 |
| 4.5 | SOFTWARE BASED HARDWARE TESTING | 138 |
| 5.0 | SUMMARY, CONCLUSIONS, AND FUTURE WORK..... | 144 |
| 5.1 | SUMMARY AND CONCLUSIONS | 144 |
| 5.2 | FUTURE WORK..... | 146 |
| 5.2.1 | Feature Additions | 146 |
| 5.2.2 | Direct Memory Access..... | 148 |
| 5.2.3 | Transformation Element Calculations | 150 |

| | | |
|-------------|--|-----|
| 5.2.4 | Using external processor over PCI express..... | 151 |
| 5.2.5 | More Parallelism in Rasterization | 152 |
| 5.2.6 | Partial Reconfigurability..... | 154 |
| APPENDIX A: | GRAPHICS PIPELINE CONTROL REGISTERS | 156 |
| APPENDIX B: | N64 CONTROLLER REGISTERS..... | 163 |
| APPENDIX C: | VHDL SOURCE CODE..... | 166 |
| C.1 | TOP LEVEL VHDL FILE..... | 166 |
| C.2 | GRAPHICS PIPELINE TOP LEVEL VHDL FILE..... | 175 |
| C.3 | MATRIX MULTIPLIER | 191 |
| C.4 | MATRIX MULTIPLIER WITH BUFFERING AND NORMILIZATION | 199 |
| C.5 | CLIPPING TREE..... | 205 |
| C.6 | OUTCODE GENERATOR | 211 |
| C.7 | CLIPPING LOGIC | 214 |
| C.8 | ABSOLUTE VALUE | 235 |
| C.9 | BRESENHAM’S ALGORITHM | 237 |
| C.10 | ZBT FRAME BUFFER..... | 250 |
| C.11 | ZBT MEMORY CONTROLLER..... | 257 |
| C.12 | ZBT PHYSICAL INTERFACE | 264 |
| C.13 | ZBT PORT INTERFACE..... | 267 |
| C.14 | ZBT ARBITER..... | 274 |
| C.15 | ZBT WIDTH CONVERSION..... | 281 |
| C.16 | ZBT MEMORY CONTROLLER PACKAGE..... | 284 |
| C.17 | DVI PHYSICAL INTERFACE..... | 287 |

| | |
|--|------------|
| C.18 VGA FRAME READER..... | 289 |
| C.19 VGA SYNC GENERATOR..... | 294 |
| C.20 VGA CONTROLLER..... | 295 |
| C.21 GRAPHICS PIPELINE TESTBENCH..... | 301 |
| APPENDIX D: C TESTCODE | 316 |
| BIBLIOGRAPHY | 330 |

LIST OF TABLES

| | |
|--|-----|
| Table 2.1: 2D Outcode Assignment Table..... | 42 |
| Table 2.2: 2D Clipping Intersection Equations..... | 43 |
| Table 2.3: 3D Parallel Projection Outcode Assignment | 47 |
| Table 2.4: 3D Perspective Projection Outcode Assignment..... | 47 |
| Table 2.5: 3D Parallel Projection Clipping Intersection Equations..... | 48 |
| Table 2.6: 3D Perspective Projection Clipping Intersection Equations..... | 48 |
| Table 3.1: 2D Outcode Assignment Table..... | 69 |
| Table 3.2: 3D Perspective Outcode Assignment Table | 69 |
| Table 3.3: 2D Clipping Intersection Equations..... | 72 |
| Table 3.4: 3D Perspective Projection Clipping Intersection Equations..... | 73 |
| Table 4.1: VGA Horizontal Timing Table..... | 127 |
| Table 4.2: VGA Vertical Timing Table | 127 |
| Table A.1: GPU Configuration Register Memory Map and Register Definition | 156 |
| Table B.1: N64 Controller Interface Memory Map and Register Definition | 163 |

LIST OF FIGURES

| | |
|---|----|
| Figure 1.1: Classic Graphics Pipeline (1) | 2 |
| Figure 1.2: GeForce 8800 CUDA Architecture (3) | 3 |
| Figure 1.3: Wireframe Modeling(1) | 8 |
| Figure 2.1: Computer Graphics Processing Pipeline(1). | 11 |
| Figure 2.2: 2D Coordinate System..... | 14 |
| Figure 2.3: 3D Cartesian Coordinate System | 14 |
| Figure 2.4: Polygon Mesh Representation(7)..... | 15 |
| Figure 2.5: Translation of a cube. | 18 |
| Figure 2.6: Scaling of a cube. | 19 |
| Figure 2.7: Differential Scaling of a cube..... | 19 |
| Figure 2.8: Right Handed Coordinate System with Rotational Angle..... | 20 |
| Figure 2.9: Rotation of a cube..... | 22 |
| Figure 2.10: Object Centered on P1..... | 23 |
| Figure 2.11: Local to World Coordinate Translation..... | 23 |
| Figure 2.12: Example Graphics Pipeline(1)..... | 26 |
| Figure 2.13: Cube in Object Coordinates | 27 |
| Figure 2.14: Cubes in World Space Coordinates..... | 28 |

| | |
|--|----|
| Figure 2.15: Viewing Point with Viewing Direction in World Coordinates | 29 |
| Figure 2.16: Viewing plane..... | 30 |
| Figure 2.17: View coordinate system. | 31 |
| Figure 2.18: View Volume | 31 |
| Figure 2.19: Viewing coordinate system defined with World coordinate system..... | 33 |
| Figure 2.20: Viewing Coordinate System..... | 34 |
| Figure 2.21: Example Graphics Pipeline(6)..... | 35 |
| Figure 2.22: Parallel Projection | 36 |
| Figure 2.23: Perspective Projection | 36 |
| Figure 2.24: Perspective Projection (7) | 37 |
| Figure 2.25: Perspective Projection (7) | 38 |
| Figure 2.26: Parallel Projection (8)..... | 40 |
| Figure 2.27: Clipping Region Definitions. | 42 |
| Figure 2.28: Illustration of 2D Cohen Sutherland Clipping. | 45 |
| Figure 2.29: Parallel Projection | 46 |
| Figure 2.30: Perspective Projection | 46 |
| Figure 2.31: Clipping to Screen Coordinates..... | 49 |
| Figure 2.32: Rasterized Line(10) | 52 |
| Figure 2.33: Bresenham’s Line Algorithm Diagram(11) | 53 |
| Figure 2.34: Line Rasterization Pseudo code. | 54 |
| Figure 2.35: Slope Octet Ranges | 55 |
| Figure 3.1: Graphics Pipeline(1)..... | 58 |
| Figure 3.2: GPU Top Level Block Diagram..... | 59 |

| | |
|---|----|
| Figure 3.3: Computer Graphic Pipeline(1) | 61 |
| Figure 3.4: 3D Graphics Pipeline..... | 62 |
| Figure 3.5: 2D Graphics Pipeline..... | 63 |
| Figure 3.6: Floating Point Matrix Multiplication Block Diagram..... | 65 |
| Figure 3.7: Clipping Logic..... | 68 |
| Figure 3.8: Outcode Generator for Clipping Logic. | 70 |
| Figure 3.9: Clipping Decision Logic State Machine | 71 |
| Figure 3.10: Cohen-Sutherland Line Clipping with outcodes | 75 |
| Figure 3.11: Edge Intersection calculator. | 76 |
| Figure 3.12: Round Robin Arbiter | 78 |
| Figure 3.13: Bresenham’s Line Rasterizer Design | 80 |
| Figure 3.14: Line Drawing State Machine..... | 82 |
| Figure 3.15: Double Buffer State Machine..... | 84 |
| Figure 3.16: Frame Buffer Interface with Frame Memory | 85 |
| Figure 3.17: Frame Reading State Machine | 86 |
| Figure 3.18: GPU Top Level Block Diagram..... | 87 |
| Figure 3.19: GPU Control Registers..... | 89 |
| Figure 4.1: ML506 Development Board (13)..... | 93 |
| Figure 4.2: GPU Top Level Design | 94 |
| Figure 4.3: GPU Top Level Implementation Block Diagram..... | 95 |
| Figure 4.4 : IEEE 754-1985 32 Bit Floating Point Number | 97 |
| Figure 4.5 : Custom 18 Bit Floating Point Number..... | 97 |
| Figure 4.6: Xilinx Coregen Floating Point Operation Selection Window..... | 98 |

| | |
|--|-----|
| Figure 4.7: Floating Point Precision Selection Window..... | 99 |
| Figure 4.8: Microblaze Core Block Diagram(17)..... | 101 |
| Figure 4.9: Multi-port Memory Interface Layout (18). | 103 |
| Figure 4.10: PLB Block Diagram (19) | 104 |
| Figure 4.11: Microblaze System..... | 105 |
| Figure 4.12: Xilinx Platform Studio’s Project Opener | 106 |
| Figure 4.13: Base System Builder Welcome Window | 107 |
| Figure 4.14: Base System Builder Board Selector..... | 108 |
| Figure 4.15: Base System Builder Processor Selector..... | 109 |
| Figure 4.16: Base System Builder Microblaze Processor Configuration Window | 110 |
| Figure 4.17: Base System Builder IO Interfaces Configuration Windows..... | 111 |
| Figure 4.18: Base System Builder Cache Setup Window..... | 112 |
| Figure 4.19: Base System Builder Software Setup Window | 113 |
| Figure 4.20: Base System Builder System Created Window | 114 |
| Figure 4.21: Original Base System without Custom Peripherals. | 115 |
| Figure 4.22: Peripheral Creation Window. | 116 |
| Figure 4.23: N64 Controller(20)..... | 117 |
| Figure 4.24: GPU Pipeline Register PLB Interface..... | 119 |
| Figure 4.25: GPU Pipeline Top Level Implementation..... | 120 |
| Figure 4.26: Matrix Multiplier Accelerator | 122 |
| Figure 4.27: Cohen Sutherland Clipping Implementation | 123 |
| Figure 4.28: Bresenham’s Line Rasterizer Design | 124 |
| Figure 4.29: Frame Buffer Interface with Frame Memory | 125 |

| | |
|--|-----|
| Figure 4.30: VGA Horizontal Sync Timing. | 126 |
| Figure 4.31: VGA Vertical Sync Timing..... | 127 |
| Figure 4.32: GPU Testbench Block Diagram..... | 129 |
| Figure 4.33: Matrix Programming in Simulation | 131 |
| Figure 4.34: Pushing Line in Simulation | 132 |
| Figure 4.35: Matrix Multiplication in Simulation..... | 132 |
| Figure 4.36: PPM unit cube. | 133 |
| Figure 4.37: Xilinx’s EDK showing processor sub-system..... | 135 |
| Figure 4.38: ISE 3D GPU hierarchy..... | 136 |
| Figure 4.39: Serial terminal output from test software. | 138 |
| Figure 4.40: 3D Graphics Processor Output..... | 139 |
| Figure 4.41: 3D Object rotation. | 140 |
| Figure 4.42: 3D Object Scaling | 140 |
| Figure 4.43: 3D Objects Translated and clipped. | 141 |
| Figure 4.44: Model Plane Rendered using GPU..... | 142 |
| Figure 4.45: Development Kit with JTAG, VGA and N64 Hardware | 143 |
| Figure 5.1: Program Memory and Object Memory Division | 149 |
| Figure 5.2: Graphics Pipeline DMA..... | 150 |
| Figure 5.3: Interleaved memory organization..... | 153 |
| Figure 5.4: Contiguous Partitioning..... | 154 |

LIST OF EQUATIONS

| | |
|--|----|
| Equation 2.1: Two Dimensional and Three Dimensional Vectors. | 12 |
| Equation 2.2: Two Dimensional and Three Dimensional Homogenous Vectors. | 13 |
| Equation 2.3: Matrix Representation of Translation, Scaling and Rotation of a Vertex | 16 |
| Equation 2.4: 3D Translation Transformation Matrix | 17 |
| Equation 2.5: 3D Scaling Matrix Transformation (1)..... | 18 |
| Equation 2.6: Z Axis Rotation Transformation Matrix (1)..... | 21 |
| Equation 2.7: Y Axis Rotation Transformation Matrix (1) | 21 |
| Equation 2.8: X Axis Rotation Transformation Matrix (1) | 21 |
| Equation 2.9: 2D Local to World Coordinate Transformation Matrix | 24 |
| Equation 2.10: 3D rotation transformation matrix..... | 25 |
| Equation 2.11: 3D Local to World Coordinate Transformation | 25 |
| Equation 2.12: View Coordinate Transformation..... | 32 |
| Equation 2.13: Similar Triangle Rations with solutions for x and y (7)..... | 39 |
| Equation 2.14: Perspective Projection Transformation Matrix (1)..... | 39 |
| Equation 2.15: Perspective Projection Transformation Matrix (1)..... | 40 |
| Equation 2.16: 2D Parametric Equations..... | 43 |
| Equation 2.17: 3D Parametric Equations..... | 47 |
| Equation 2.18: General Screen Transformation Matrix..... | 50 |

| | |
|---|----|
| Equation 2.19: General Screen Transformation Matrix..... | 51 |
| Equation 2.20: General line equation through two endpoints..... | 52 |
| Equation 2.21: Line equation solved for y..... | 53 |
| Equation 3.1: Matrix Multiplication(6)..... | 65 |
| Equation 4.1: Floating point calculation..... | 97 |

PREFACE

I would like to thank Dr. Cain for the opportunity to pursue the project and for all the guidance he has provided through this ordeal. Also, thanks to my committee of Dr. Jones and Dr. Chang for their input and guidance. I would also like to thank my family for all their help and support while working on this project. Lastly, I would like to dedicate this work to my loving wife Marcie whose support was instrumental in my ability to complete this thesis.

1.0 INTRODUCTION

1.1 OVERVIEW

Real-time computer graphics hardware designs have massively changed over the years. In early stages of graphics processing, such tasks were delegated to a Central Processing Unit (CPU). CPUs are highly flexible programmable devices that excel at performing sequential tasks. Despite CPUs flexibility and usefulness, they do not do as well at executing tasks which can take advantage of parallelism such as graphic processing. Graphic processing requires many complex mathematical operations to be performed in real-time. In the late 90s and early 2000s demand for more complex graphic processing increased. This has led to custom hardware solutions which take advantage of pipelining and the massive parallelism inherent in graphics processing. Today, these custom hardware solutions are implemented using Application Specific Integrated Circuits or ASICs. Until recently ASIC based graphics processors used fixed pixel pipelines architectures. Today, many modern graphics processors have switched to programmable vertex and pixel processors greatly increasing the graphics processors performance and flexibility.

Until this recent change, graphics processing used the more traditional fixed pipeline shown in Figure 1.1.

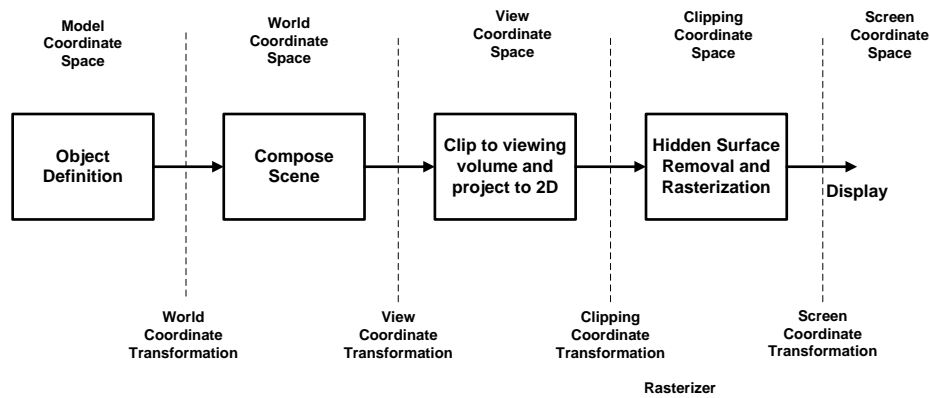


Figure 1.1: Classic Graphics Pipeline (1)

This pipeline architecture uses dedicated hardware blocks to render a 3D scene. A pipeline such as this operates on three dimensional vertices. Vertices are 3D points which are grouped together to form complex objects. The object definition step in Figure 1.1 is storage space for groups of vertices that form objects. Graphics processors are often paired with central processing units (CPUs). In most systems, graphics objects are stored in CPU memory and then passed to the pipeline by the CPU itself. These objects are then passed to the compose scene block where coordinate transformations and lighting calculations are performed. Coordinate transformations are necessary to place 3D objects into a 3D virtual world. In addition, clipping and projections are necessary to display the 3D virtual world onto a 2D screen. These 3D transformations, 2D projections and other functions are discussed in more detail in Section 2.0 . Next, in the hidden surface removal and rasterization stage, objects are converted to pixels (called rasterization), shaded, anti-aliased and then textured. Pixels are discrete points of color on a 2D screen and the process of rasterizing 3D objects into pixels will be discussed further in Section 3.0 . Lastly, these pixels are stored into the graphic's pipeline's local memory (called the frame buffer) in the hidden surface removal and rasterization stage to create a 2D frame of pixel data. This pixel data is then read from the frame buffer and driven to a human display interface.

The graphics pipeline has basically included these same stages for the past two decades with successive improvements over the years. The largest improvements have been finding ways to increase the memory bandwidth of the frame buffer through parallelism and memory interleaving. In addition, improvements have been made by brute force simply by creating many graphics pipelines such as the one above which operate on a 3D scene in parallel.

Today, graphics processing is taking a radical new direction. An example of this is the Geforce 8800 GS graphics processor from NVIDIA (2). The 8800 uses the Computer Unified Device Architecture (CUDA) which provides a multiprocessor based solution for graphics processing. The architecture is shown in Figure 1.2.

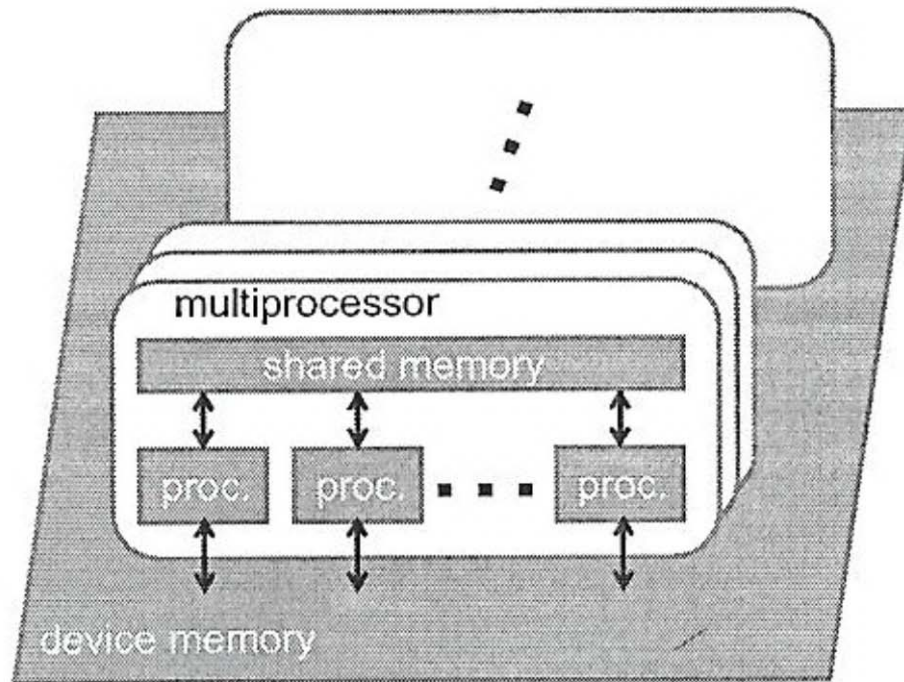


Figure 1.2: GeForce 8800 CUDA Architecture (3)

The CUDA architecture differs from the classic graphics pipeline in that instead of set pipelined hardware blocks, many specialized scalar processors are used. The Geforce 8800 CUDA architecture can perform tens of thousands of threads concurrently. Using the CUDA

architecture, each scalar processor can be tasked with vertex, pixel or geometric calculations to perform any function implemented in the classic pipeline.

The CUDA architecture in Figure 1.2 consists of 16 multiprocessors which each share 768 MB of system memory. Within each of the 16 multiprocessors, there are eight scalar processors which share a local 16 KB cache. In total there are 128 scalar processor (16 multiprocessor * 8 scalar processors) each running at 1.35 GHz. These scalar processors specialize at executing matrix-vector multiplication efficiently. Matrix-vector multiplication, as will be seen later in Section 2.1.4, is a very common operation in graphics processing. Using architectures such as this, today's graphics processors are capable of rendering very complex 3D worlds in real-time.

Currently, the majority of graphics processing units (GPUs), such as the state of the art Geforce 8800 GS, are designed and implemented as Application Specific Integrated Circuits (ASICs). ASICs can be a good choice because of a GPU's high speed performance requirements. However, ASICs are very expensive to develop and manufacture due the high cost of photomasks as well as the massive up front verification effort needed. It is not uncommon for ASICs to require multiple revisions to iron out all the flaws of a particular design. With the development of each photomask costing in the millions, development cost can escalate quickly.

In contrast, Field Programmable Gate Arrays (FPGAs) are devices that contain programmable logic blocks with a programmable interconnected fabric to connect these various blocks. FPGAs can eliminate many of the problems associated with ASIC design. FPGAs are reconfigurable devices and due to this fact do not require the large scale up front verification of ASICs. In addition, FPGA's reconfigurability eliminates the expensive photomask cost penalty

that ASIC design defects would accrue through design defects. These facts make FPGAs ideal for digital system prototyping.

FPGAs also present a variety of additional advantages. For instance, if an FPGA based design is found to have a design defect in the field, firmware updates allow the defects to be fixed without massive hardware recalls or hardware replacement. In addition, new features can be added based on new customer needs. Lastly, new techniques such as on the fly reconfigurability enable parts of the FPGA to be reconfigured to implement different functions using the same physical hardware.

Some disadvantages of FPGAs is that they cost more per unit as compared to ASICs (eliminating the upfront ASIC development costs) as well as the fact that ASICs generally outperform FPGAs in terms of performance and density. Despite these disadvantages, FPGAs have now begun to bridge the performance gap coming close enough in terms of performance of ASICs to be used in many applications where ASICs were traditionally used. Xilinx, a leading FPGA vendor, has recently released the Virtex 5 SXT device (4). This device is a high density high performance FPGA with 32,640 look up tables, 32,640 registers and a maximum clock speed of 550 MHz. Altera, a competing FPGA vendor, offers similar performing products to the Virtex 5.

These FPGA specs, while impressive by FPGA standards, make developing graphics processors that can compete with today's state of the art ASIC based graphics processors, such as the Geforce 8800, impractical. The sheer size of the Geforce 8800 (coming in at 754 million transistors (2)) is very unlikely to be implemented in such a way as to fit into the limited number of logic elements an FPGA provides. Another problem is the clock speeds of the scalar processors, where the Geforce 8800 has its processing cores running at 1.35Ghz, far above what

today's FPGA can achieve. Although the state of the art is currently impractical, many legacy GPU designs have core clock speeds in the 200 to 300 MHz range. GPU such as these also use the classic GPU architecture shown in Figure 1.1. This architecture is simpler to conceptualize due to the fact that each stage is a well defined custom hardware structure as opposed to a fully functional scalar processor as in the Geforce 8800.

Taking the advantages of FPGAs into consideration, an FPGA based graphics processor using the architecture in Figure 1.1 could be used in many applications. For instance, a FPGA based graphics processor could be use in many embedded applications where a fully featured graphics processor is not needed. One such field is image processing. In image processing many autonomous robots use combinations of cameras for visual data and things such as LADAR scanners to acquire 3D depth information. LADAR is an acronym for Laser Radar. This technique uses echoed laser beams as opposed to sound waves to get a 3D information from a surface. FPGAs could be used to implement basic graphics processing logic which could map the camera's 2D visual information to the LADAR's 3D depth information. This could be used by robots to record a very accurate landscape of what it has scene in the past. The beauty of FPGAs is that the image processing logic, graphics processing logic, and other general purpose logic could all be integrated into a single system on a chip FPGA design. These are just some of the possible applications graphics processing using FPGAs.

Another possible application is legacy ASIC based graphics processor replacement. FPGA based graphics processors could be used to replace legacy ASIC graphics processors where the size and performace of the legacy device makes it feasible. FPGA designs have the advantage of being field upgradable as well as on the fly reconfigurable providing several advantages over the standard ASIC based legacy graphics processors as well. These legacy

graphics processors could also be integrated in system on chip designs where graphics processing as well as other functionality is needed on a single chip.

Lastly, FPGA base graphics processors could be used for a rapid prototyping where certain features of more complex graphics processor could be tested in physical hardware. This approach could minimize risk in many graphics processing designs. Instead of relying only on simulators the hardware architecture could also be tested in real hardware on an FPGA for basic functionality. This could save millions of dollars in GPU ASIC refabrication due to additional bugs found in the FPGA debugging phase possibly preventing several stages of costly chip refabrication. In order to determine if such applications are feasible in an FPGA first a basic prototype using FPGA technology must be designed and tested.

1.2 STATEMENT OF THE PROBLEM

The overview above discussed the state of the art of graphics processing. Two different architectures used today were discussed, the classic fixed pipeline and the multiprocessor approach. The overview made a case that the fixed graphics pipeline is the most ideal for implementation in an FPGA. In addition, the overview above showed certain applications could take advantage of an FPGAs based graphics processing implementation as opposed to using ASICs for development.

The objective of this thesis is to test the feasibility of implementing complex graphics processing functions on modern FPGA devices. In order to accomplish this, this thesis presents an FPGA based graphics processor design and implementation that establishes the feasibility of a

complex set of graphics processing functions. From this FPGA prototype many conclusions can be drawn about the performance, power and cost of an FPGA based GPU.

The FPGA prototype requirements include the features listed below.

- Graphics processing engine capable of rendering in both 2D and 3D.
- Rendering of wireframe objects.
- Support for a free roaming point of view (or camera) in 3D.
- Support standard display interfaces.

The features above constitute a fully functional graphics system. One such feature, the free roaming camera allows for manipulation of objects within a virtual world. These objects can be rotated, scaled or moved within this virtual world. The objects can then be displayed onto a 2D computer display. For this proof of concept design, the objects to be displayed will be wireframe based. Wire frames models are defined only by the edges of a physical object. Some examples of wireframes are shown in Figure 1.3.



Figure 1.3: Wireframe Modeling(1)

Wire frame modeling is conceptually simple to implement compared textured graphics. The advantage of wire frames are the high frame rates they allow due to the small pixel fill rates needed. Using wireframes in conjunction with clipping and hidden surface removal will allow this first pass system to run complex scenes in real time at 60 frames per second. The use of

wire frames will be sufficient for proof of concept while advanced features such as surface texturing, lighting and shading could be implemented with successive additions to the design.

This thesis presents the work performed to realize an FPGA based graphics processor prototype. This document first must define requirements which are subset of graphics processing functions to be implemented. Once identified and described, the functions must be designed to run efficiently within an FPGA device. Given the design requirements, an FPGA must be selected that has the logic resource and performance to implement these proposed graphics functions. Following implementation, the FPGA based graphics processor must then be fully verified and tested for functional correctness as well as have its area, power and performance measured. From the graphics processor area and performance numbers, various conclusions can be drawn on what graphics processing features are currently possible to implement in today's FPGA devices as well as what advantages an FPGA based designs presents over an ASIC based design.

1.3 OUTLINE

The next section, Section 2.0 , is an introduction to computer graphics. It gives the reader all the necessary mathematical background information on the features planned for design and implementation in the following sections. In the third section the functional requirements of the prototype as well as how the prototype has been designed to meet these requirements are presented. Section 4.0 shows the detailed implementation, verification and testing of the system presented in Section 3.0 . Finally, Section 5.0 gives the conclusions on the performance of the designed graphics processor and shows some ideas for future work.

2.0 COMPUTER GRAPHICS RENDERING

Computer graphics rendering is the process of displaying an image on a computer display given a list of geometrically defined objects. These objects can be defined in either two-dimensional space (2D) or three-dimensional space (3D). This 2D or 3D space is referred to as a virtual world.

Though virtual world descriptions are three-dimensional, in contrast computer displays are two dimensional. In order to display a three dimensional image on a two dimensional computer display, a two dimensional projection of the three dimensional image is needed. In addition, clipping to the borders of portions of the world viewable on the screen may be necessary to increase performance. Given this clipped two dimensional description, the virtual world can then be mapped to the computer display.

Today, most computer displays are raster based. Raster graphics (5) are a rectangular grid of points of color (know as pixels) used to display an image. The two dimensional projected objects defined in vector format must be quantized and converted to a raster image. This process is known as rasterization. Rasterization involves determining which pixels in the raster display are affected by each object in the virtual scene. This process involves functions such as line drawing. Rasterization will be discussed later.

Figure 2.1 shows a high level description of a graphics processing pipeline. In the pipeline, all the consecutive steps needed to display a list of abstract 2D or 3D objects onto a 2D computer display are shown.

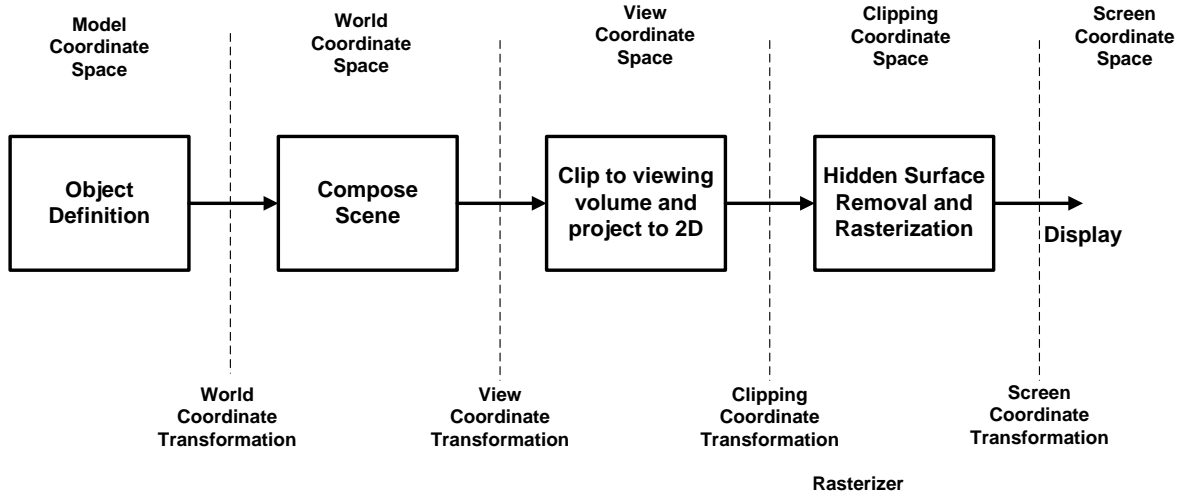


Figure 2.1: Computer Graphics Processing Pipeline(1).

As can be seen in the figure above, computer graphics involves converting from one coordinate system to another. Starting at the left, model coordinates are where each object is stored in its own coordinate system. Objects in this coordinates system usually have a control point which is used to move the model around the world coordinate system relative to the control point. The next coordinate system is the world coordinate system. The world coordinate system is the model of the actual 3D or 2D world where various objects defined in model coordinate space can coexist. Next is the view coordinate system. This system can be thought of as the world coordinate system viewed from a single point in space (like a camera). Next follows the clipping space, where the world is constrained into a finite viewing volume and then projected on a 2D plane. Lastly, the screen coordinate system is the rasterized version of the 2D projection. This final coordinate system corresponds to the physical display and is driven to the display device.

The following section will start with a brief overview of the mathematical principles needed to implement the requirements of the graphics pipeline presented in Section 1.2. This is followed by a detailed discussion of all the steps taken by the graphics pipeline to convert a list of graphics objects to pixels on the screen.

2.1 MATHEMATICS OVERVIEW

Transformations are tools that can be used to manipulate 2D and 3D objects in a virtual world. They can be used to move an object within the virtual world and project the 3D virtual world onto a 2D plane. This section introduces the notion of a vector as well as how vectors can be grouped together to form graphics objects. These objects can then be transformed using basic 2D and 3D affine geometric transformations discussed here.

2.1.1 Homogenous vectors

In computer graphics 2D and 3D objects are defined in respect to a virtual world. This virtual world is nothing more than a mathematical representation of a world's geometry. Objects can be defined in many ways, but are most commonly defined as a group of vertices. These vertices are represented as vectors as shown in Equation 2.1.

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

Equation 2.1: Two Dimensional and Three Dimensional Vectors.

The x, y and z value of each vector describes an object's position within a coordinate system. Coordinate systems are discussed in Section 2.1.2.

In computer graphics, vertices are often described in homogenous form(6). Homogenous representation can allow affine geometric transformations to be easily represented by a single matrix multiplication. In this representation, coordinates are the same as before except an additional w coordinate is added to the vector as shown in Equation 2.2.

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

Equation 2.2: Two Dimensional and Three Dimensional Homogenous Vectors.

The need for homogenous vectors will be made apparent in the affine geometric transformation Section 2.1.4.

2.1.2 Coordinate System

Coordinate systems are used to define a virtual space in computer graphics. The relative positions of an object as well as its dimensions are all defined within these coordinate systems. 2D and 3D coordinate systems vary by the number of axes each systems has.

For instance, a coordinate system in two dimensions is defined by two axes at right angles to each other forming a xy-plane. A grid on each axis defines uniform distances from center of the coordinate system know as the origin. The basic 2D coordinate system is shown in Figure 2.2

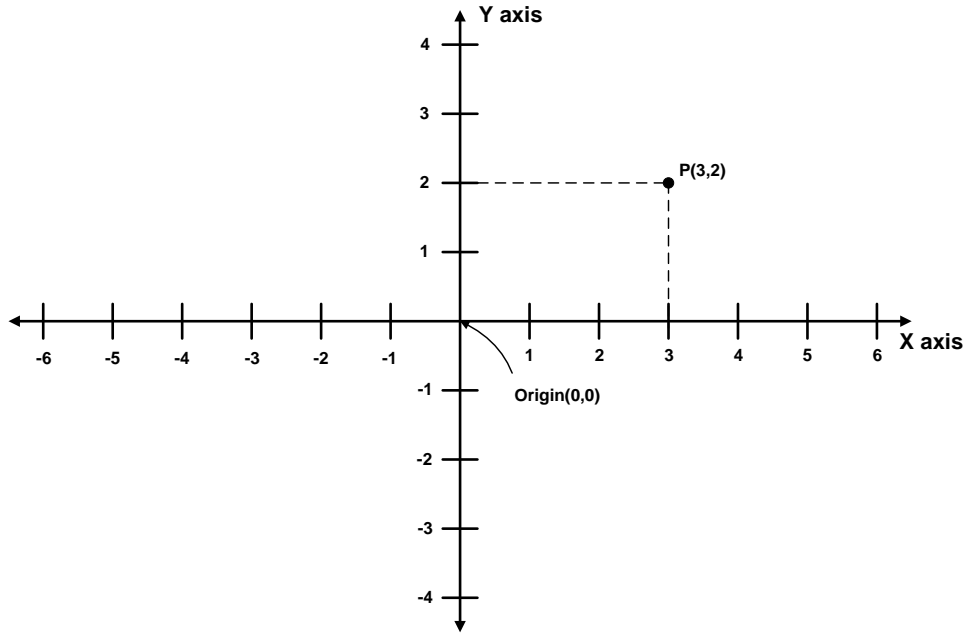


Figure 2.2: 2D Coordinate System

In Figure 2.2 the 2D vector $P(3,2)$ is defined.

The three dimensional Cartesian coordinate system is shown in Figure 2.3.

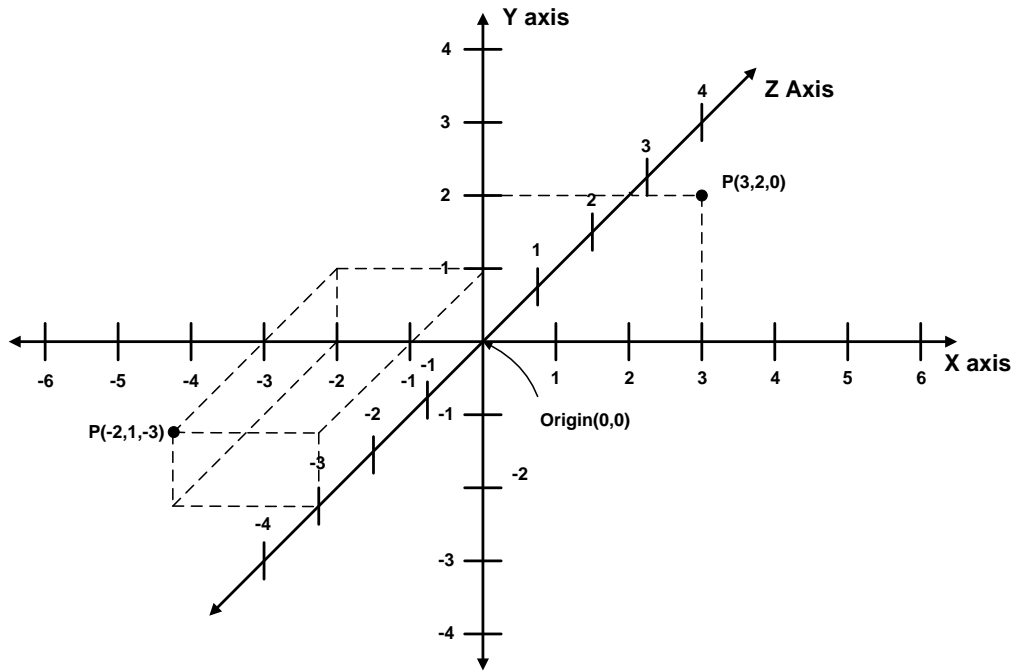


Figure 2.3: 3D Cartesian Coordinate System

Note that the original vector in the 2D coordinate system of Figure 2.2 is now located at P(3,2,0) where the z coordinate is zero. Another 3D vector at P(-2,1,-3) is also defined in the coordinate system.

Vector objects can be used to describe a group of connected points within a virtual world defined by a coordinate system. These connected points are known as a polygon mesh. These polygon mesh objects can then be moved with ease using affine geometric transformations. Polygon meshes and their use with affine geometric transformations are discussed in the next subsections.

2.1.3 Object Representation

Today most commercial graphics processing units use a polygon mesh(7) to represent 2D or 3D objects. A polygon mesh is a collection of vertices joined together to form the polygons of an object. Figure 2.4 below shows an example of a four point object defined as a polygon mesh. V_1, V_2, V_3 and V_4 define the vertices that make up an object while P_1 and P_2 are the polygons defined by this list of vertices.

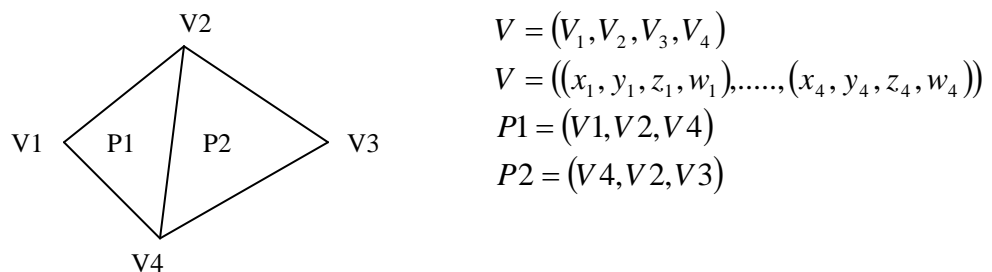


Figure 2.4: Polygon Mesh Representation(7)

The list of vertices, such as V_1, V_2, V_3 and V_4 shown in Figure 2.4 can be converted from one coordinate system to another quite easily using a linear transformation matrix. The next section

describes how these objects can be scaled, rotated or translated using affine geometric transformations.

2.1.4 Affine Geometric Transformations

Affine geometric transformations can be used to move objects around in a virtual environment, convert between coordinate systems, and implement projections of 3D images on to a 2D plane. This section gives the background on two dimensional and three dimensional affine geometric transformations. An affine transformation is any transformation involving scaling, rotation, or translation from one coordinate system to another.

A set of vectors (or points) which define an object can be transformed into another set of points by an affine geometric transformation. Matrix notation is used in computer graphics to describe an affine geometric transformation. The convention in computer graphics is to have a vector V as a column vector which is multiplied by a transformation matrix.

Using this notation, any vector V can be transformed to a new vector V' . The translation, scaling and rotation transformations calculations are shown below.

$$V' = D + V$$

$$V' = S * V$$

$$V' = R * V$$

Equation 2.3: Matrix Representation of Translation, Scaling and Rotation of a Vertex

D is the translation vector and S and R are the scaling and rotation matrices. These three basic operations are the most common in computer graphics and can be combined to perform very complex functions (for example projection of a 3D image onto a 2D plane). The next several sections present each basic transformation individually.

2.1.4.1 Translation Transformation

Translation is the process of moving vertices from one point in a coordinate system to another by adding a translation amount to each vertex. The algebraic and matrix representation for 3D translation are shown in Equation 2.4 (for 2D removes all z components).

$$V' = D(d_x, d_y, d_z) \cdot V$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x + d_x \\ y' = y + d_y \\ z' = z + d_z \\ 1 \end{bmatrix}$$

Equation 2.4: 3D Translation Transformation Matrix

Note that in Equation 2.3 that the new vector is calculated by using matrix addition while in Equation 2.4 that the new vector uses multiplication. This is where the importance of homogenous coordinates discussed in Section 2.1.1 becomes apparent. Adding the addition homogenous coordinate $w=1$ allows translation to be expressed as a matrix multiplication instead of a matrix addition. This gives all three operations (scaling, rotation, and translation) a uniform method of calculation. This becomes important in maximizing hardware efficiency both due to the fact that only multiplication is needed and that, as will be shown later, multiple matrix transformations can be multiplied together to be combined into one transformation matrix.

Once again D is the translation matrix and defines the displacement of each component of V . Translation is more concisely explained in Figure 2.5 which shows a cube defined at the origin being moved by the translation values $[D_x, D_y, D_z] = [3, 2, 1]$.

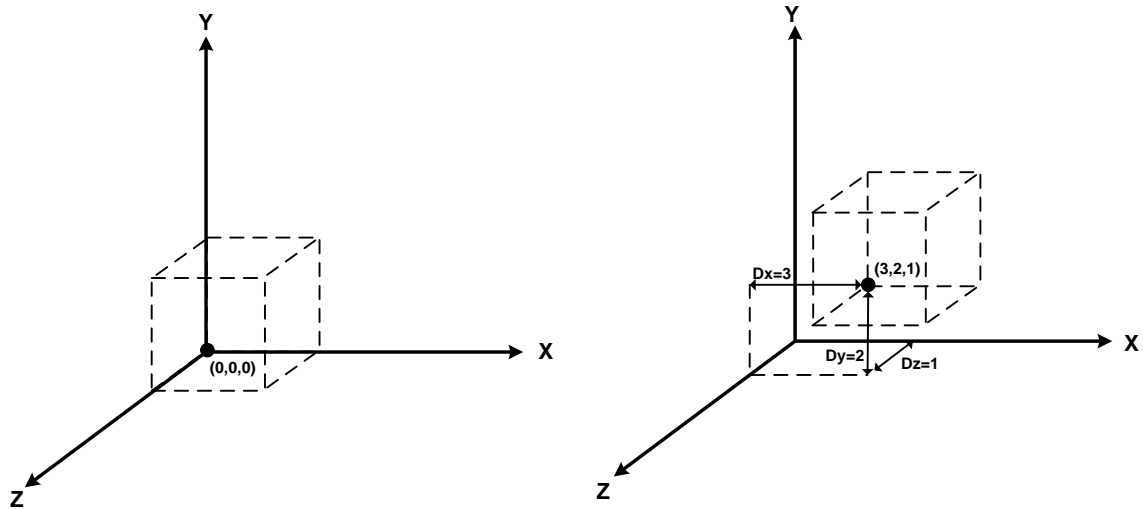


Figure 2.5: Translation of a cube.

2.1.4.2 Scaling Transformation

Scaling is another form of transformation operation. It stretches or compresses vertices within a coordinate system. This is done by multiplying each vector V by scaling factors S_x , S_y , and S_z . The algebraic and matrix representations are shown below (for 2D remove all the z components).

$$V' = S(s_x, s_y, s_z) \cdot V$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = xs_x \\ y' = ys_y \\ z' = zs_z \\ 1 \end{bmatrix}$$

Equation 2.5: 3D Scaling Matrix Transformation (1)

S is the scaling matrix and based on its values the vector V can be either expanded or compressed. This is shown in the example below in Figure 2.6. This particular example shows a cube defined at the origin being scaled by scaling factors $[S_x, S_y, S_z] = [2, 2, 2]$.

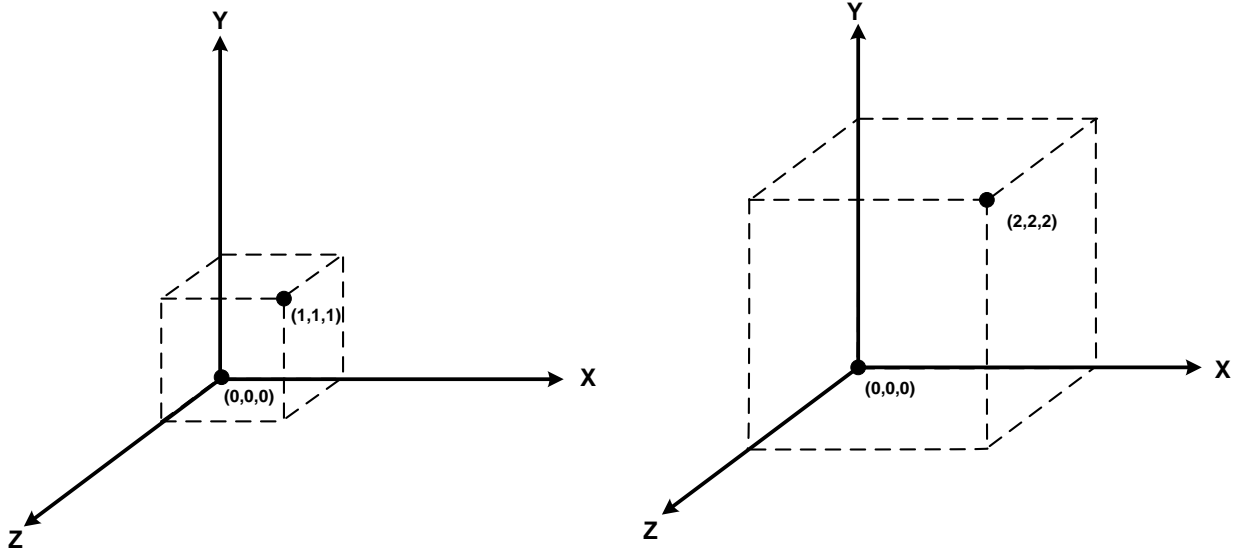


Figure 2.6: Scaling of a cube.

Note that above is an example of uniform scaling. Uniform scaling is where $S_x=S_y=S_z$. If any of the scaling factors S_x , S_y or S_z are not equal the scaling is said to be differential. Differential scaling affects the relative proportions of the object. This can be shown below in Figure 2.7 with the same cube but this time with scaling factors $[S_x,S_y,S_z] = [1/2,2,3]$.

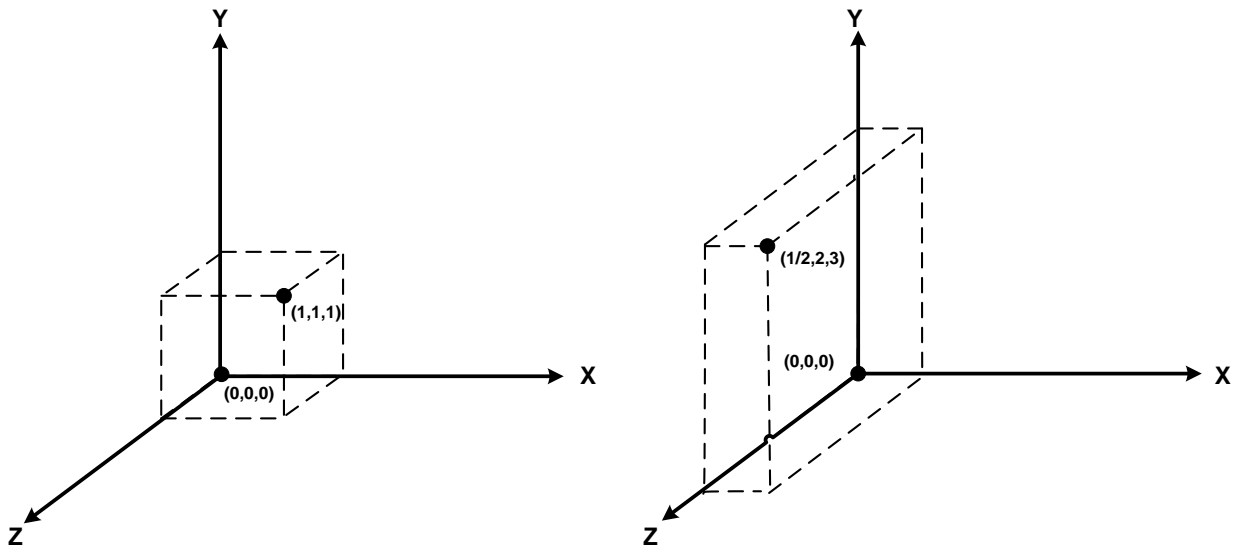


Figure 2.7: Differential Scaling of a cube.

Note that the cube, in difference to the cube scaled by factors $[2,2,2]$, has been stretched more in the Z direction, less in the X direction and the same amount in the Y direction. Differential

scaling is often used when mapping to screen coordinates of a display device that does not have the same number of horizontal and vertical lines. For instance a 640x480 VGA display.

2.1.4.3 Rotation Transformation

Rotation is revolving a vector around the origin of a coordinate system. This is accomplished by multiplying each vector by a rotation matrix. In three dimensions there are three different angles for which an object can be rotated. Each angle component is show in the Figure 2.8 below.

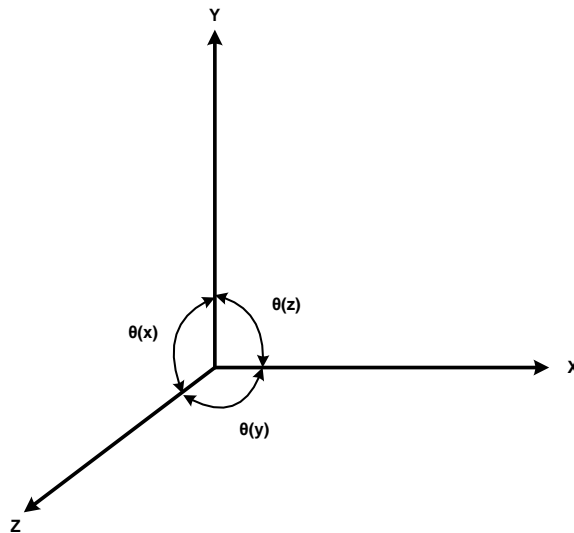


Figure 2.8: Right Handed Coordinate System with Rotational Angle.

The three transformation matrices below can be used to rotate an object around either the x, y or z axis. These equations are shown below. In each case R represents the rotation matrix and V is the vertex being rotated.

$$V' = R(\theta_z) \cdot V$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x \cos(\theta_z) - y \sin(\theta_z) \\ y' = x \sin(\theta_z) + y \cos(\theta_z) \\ z' = z \\ 1 \end{bmatrix}$$

Equation 2.6: Z Axis Rotation Transformation Matrix (1)

$$V' = R(\theta_y) \cdot V$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x \cos(\theta_y) + z \sin(\theta_y) \\ y' = y \\ z' = x \sin(\theta_y) - z \cos(\theta_y) \\ 1 \end{bmatrix}$$

Equation 2.7: Y Axis Rotation Transformation Matrix (1)

$$V' = R(\theta_x) \cdot V$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' = x \\ y' = y \cos(\theta_x) - z \sin(\theta_x) \\ z' = y \sin(\theta_x) + z \cos(\theta_x) \\ 1 \end{bmatrix}$$

Equation 2.8: X Axis Rotation Transformation Matrix (1)

Figure 2.9 shows an example of a cube rotated by $\theta_y = -45$ degrees. Note that both θ_x and θ_z can be rotated similarly to the above figure. The next section shows how to compose the various transformations into a single matrix transformation.

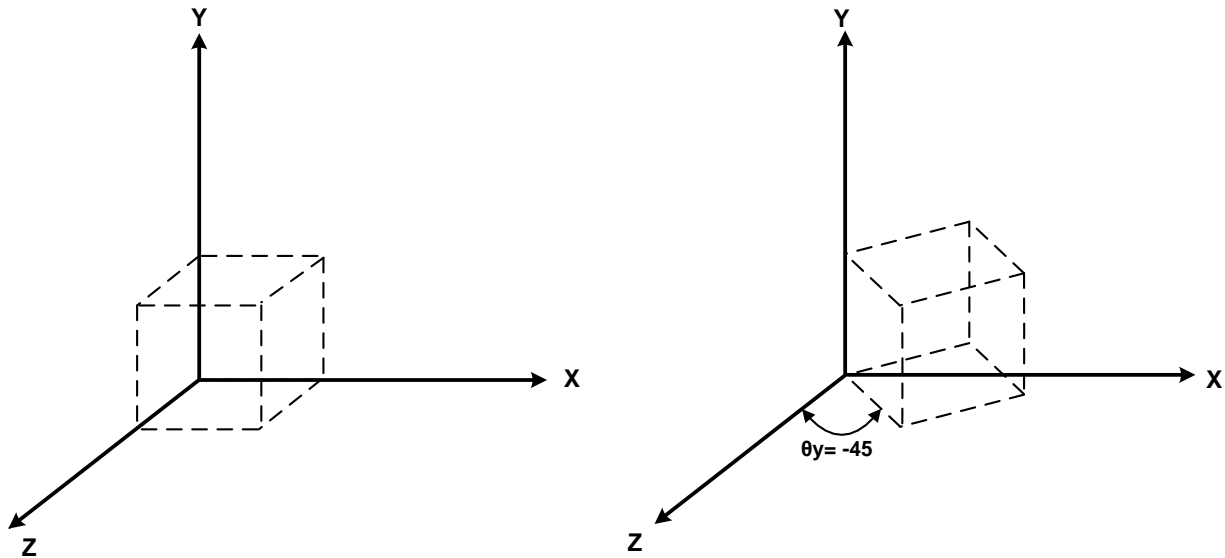


Figure 2.9: Rotation of a cube.

2.1.4.4 Transformation Compositions

Individual scaling, rotation and translation geometric transformation matrices can be combined to form just about any transformation function that is needed in computer graphics. Though these matrices can be cascaded and multiplied one at a time to achieve the desired result, matrix multiplication is a resource intensive function to implement in hardware. In order to minimize the amount of logic needed to implement a given affine geometric transformation, these matrices can be combined through multiplication to form a single matrix. This multiplication of matrices by one another is known as composition. The above scaling, rotation and translation matrices can be composed together by simple multiplication of the matrices. The main reasoning behind composing transformation matrices is to gain efficiency by applying single transformation matrices to vertices instead of applying a series of transformations. An example of this is the transforming of an object in its local coordinate system, to an object in the world coordinate system. In order to illustrate this consider the 2D object in Figure 2.10 below with a reference point on the house defined by the point P1.

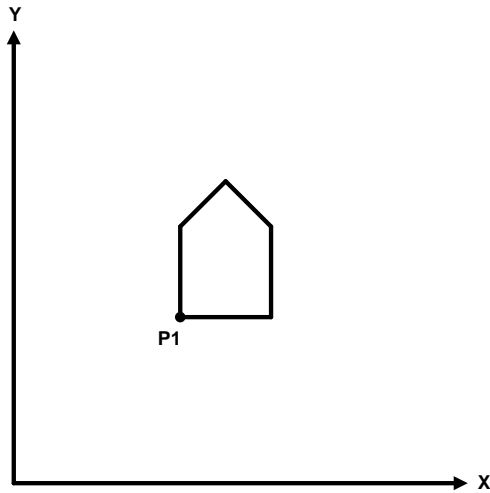


Figure 2.10: Object Centered on P1

Converting an object from object coordinates to world coordinates can involve translation, scaling, and rotation. In order to properly scale and rotate an object, first the object's reference point must be aligned to the origin of the object coordinate system. Next, the object is transformed through a scaling matrix S . Rotation can follow by transforming the resulting scaled coordinates through the rotation matrix R . Lastly, the scaled, rotated object can be translated to its position in the world coordinate system by a translation matrix T . The sequence is shown in Figure 2.11 below.

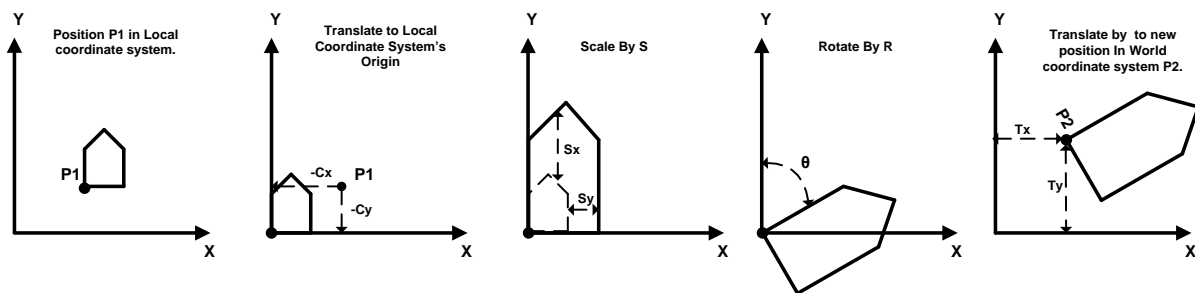


Figure 2.11: Local to World Coordinate Translation.

To do the following matrix transformation from local to world coordinates requires four matrix transformations (translation to center, scaling, rotation and translation to world coordinate

position). In order to be more efficient, these four translation matrices can be composed into a single matrix by simply multiplying the four matrices together. This process is shown in Equation 2.9 below where M is the resulting composed matrix.

$$M = D(t_x, t_y) \cdot R(\theta) \cdot S(s_x, s_y) \cdot D(-c_x, -c_y)$$

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} s_x \cos(\theta) & -s_y \sin(\theta) & t_x - c_x s_x \cos(\theta) + c_y s_y \sin(\theta) \\ s_x \sin(\theta) & s_y \cos(\theta) & t_y - c_y s_y \sin(\theta) - c_x s_x \cos(\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 2.9: 2D Local to World Coordinate Transformation Matrix

Similarly, a matrix for 3D local to world coordinate transformation matrix can be composed. 3D involves three discrete angles as shown previously in Figure 2.8. In order to create the composed matrix for transforming from object coordinates to world coordinates, first the three rotation matrices must be composed as shown below.

$$R(\theta) = R(\theta_z) \cdot R(\theta_x) \cdot R(\theta_y)$$

$$R(\theta) = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ \cos(\theta_x) & -\sin(\theta_x) & 0 & 0 \\ \sin(\theta_x) & \cos(\theta_x) & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta_y) & \sin(\theta_y) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_x) & \cos(\theta_x) & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R(\theta) = \begin{bmatrix} -\sin(\theta_z)\sin(\theta_x)\sin(\theta_y) + \cos(\theta_z)\cos(\theta_y) & -\sin(\theta_z)\cos(\theta_x) & \sin(\theta_z)\sin(\theta_x)\cos(\theta_y) + \cos(\theta_z)\sin(\theta_y) & 0 \\ \cos(\theta_z)\sin(\theta_x)\sin(\theta_y) + \cos(\theta_z)\cos(\theta_y) & \cos(\theta_z)\cos(\theta_x) & -\cos(\theta_z)\sin(\theta_x)\cos(\theta_y) + \sin(\theta_z)\sin(\theta_y) & 0 \\ -\sin(\theta_z)\cos(\theta_x) & \sin(\theta_x) & \cos(\theta_x)\cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R(\theta) = \begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & r_{12} & 0 \\ r_{20} & r_{21} & r_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 2.10: 3D rotation transformation matrix.

In the last step in Equation 2.10 the complex elements of the R matrix composed of sine and cosine are referenced by the values $r_{row,col}$. This rotation will be used in all calculations using $R(\theta)$ in future calculations in this thesis. Taking the result for the 3D rotational transformation R and multiplying it by the center translation, scaling, and world translation yield the overall result shown below in Equation 2.11.

$$M = T(t_x, t_y, t_z) \cdot R(\theta) \cdot S(s_x, s_y, s_z) \cdot T(-c_x, -c_y, -c_z)$$

$$M = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & r_{12} & 0 \\ r_{20} & r_{21} & r_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} s_x r_{00} & s_y r_{01} & s_z r_{02} & -c_x s_x r_{00} - c_y s_y r_{01} - c_z s_z r_{02} + t_x \\ s_x r_{10} & s_y r_{11} & s_z r_{12} & -c_x s_x r_{10} - c_y s_y r_{11} - c_z s_z r_{12} + t_y \\ s_x r_{20} & s_y r_{21} & s_z r_{22} & -c_x s_x r_{20} - c_y s_y r_{21} - c_z s_z r_{22} + t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 2.11: 3D Local to World Coordinate Transformation

The two matrices defined in Equation 2.9 and Equation 2.11 can be used to transform any arbitrary point in 2D or 3D respectively from one coordinate system to another. The next sections use the affine geometric transformations discussed here to transfer objects to different coordinate systems.

2.2 THE GRAPHICS PIPELINE

Recall the graphics pipeline shown again in Figure 2.12 below.

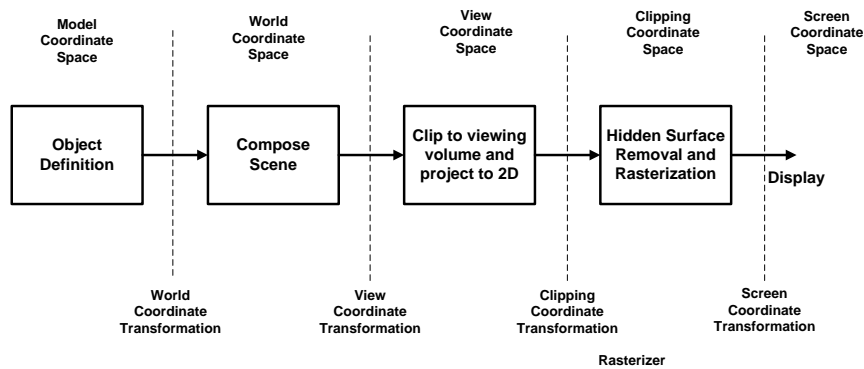


Figure 2.12: Example Graphics Pipeline(1)

In a graphics pipeline, objects are stored in a suitable format that allows easy execution of affine geometric transformations. The polygon meshes discussed in Section 2.1.3, are one such modeling technique that allows objects to be converted to different coordinate systems quite easily. Using these mesh models, it will be shown that objects can be converted to a variety of coordinate systems for processing in the graphics pipeline. The end result of the pipeline being a two-dimensional projected image on a raster display. The rest of this section goes through each stage of the graphics pipeline and explains all the operations that take place in each stage.

2.2.1 Object Definition

In the object definition stage of the graphics pipeline, polygon mesh objects are each stored in their own local coordinate system (also known as model coordinates). Figure 2.13 shows an example of a unit cube defined in the model coordinate system. Note that one of its corners is lined up around at object coordinates origin. This origin is known as the object's control point. All scaling, rotation and translation of this object will be done centered around that point.

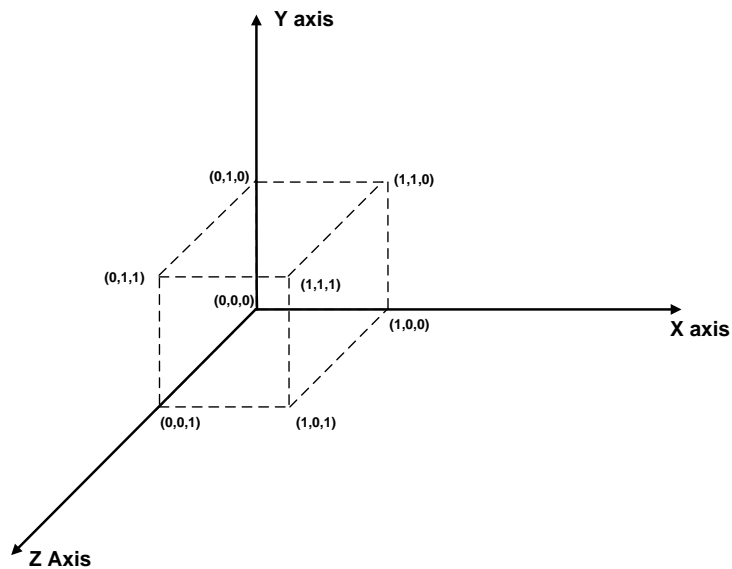


Figure 2.13: Cube in Object Coordinates

Given objects such as these, a geometric transformation can be used to convert objects in model coordinates to the world coordinate system. To make this transformation, each object is associated with scaling, rotation and translation factors. These factors are the same as the S, R and T matrices defined in Equation 2.9 and Equation 2.11. Using these equations a 2D or 3D object in model coordinates can be placed with any position or orientation within the world coordinate system.

2.2.2 Scene Composition using World Coordinate Transformation

Objects modeled in their own independent model coordinate system can then be placed in the world coordinate system by a simple geometric transformation. The world coordinate system represents each object's relative position to one another in the virtual world. Figure 2.14 below shows two unit cubes modeled from their model coordinate description in Figure 2.13 placed in world space at coordinates $(4,2,1)$ and $(1,0,4)$.

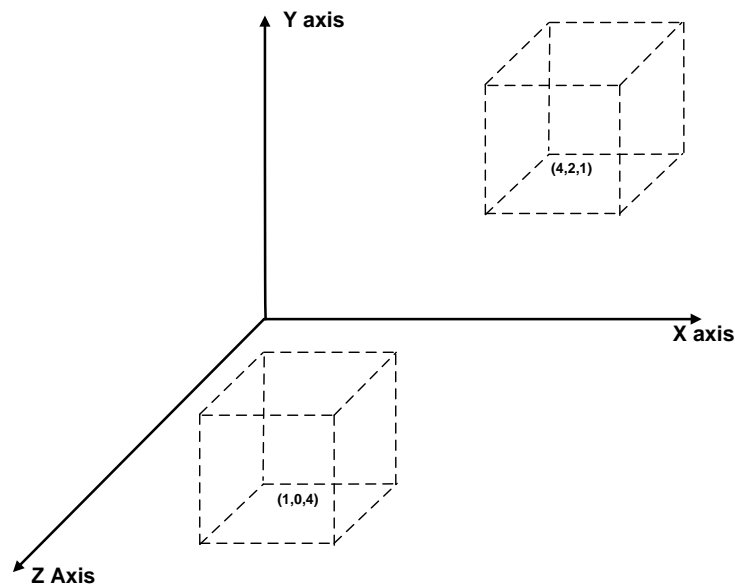


Figure 2.14: Cubes in World Space Coordinates

The geometric transformation for converting an object from model coordinates to world coordinates is identical to the matrix transformations in Equation 2.9 and Equation 2.11 for 2D and 3D objects respectively.

In this stage of the pipeline, objects can be animated by varying the translation, scaling and rotation values in the world transformation matrix. Varying over time will give the viewer a sense of an object moving on the screen. In this stage, lighting and shading can be implemented

if desired. These features will not be discussed here because the requirements for this design only necessitate wireframe graphics where lighting and shading are not necessary.

2.2.3 View Coordinates and the View Transformation

In computer graphics, viewing coordinates or the viewing coordinate system can be conceptualized as a camera pointed in a defined direction. The camera is positioned at a point within the world coordinate system called the viewing reference point (VRP) and is given a direction called the view plane normal (VPN). Figure 2.15 below shows the system in world coordinates.

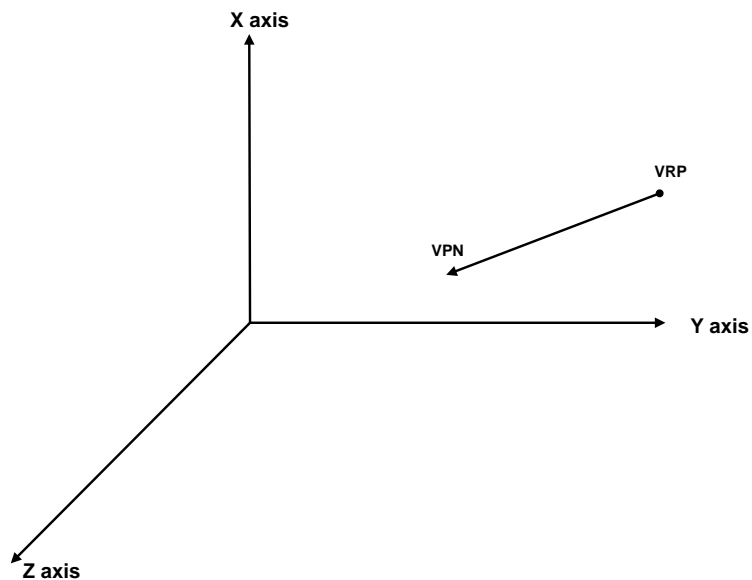


Figure 2.15: Viewing Point with Viewing Direction in World Coordinates

Using the VRP and VPN, the world coordinates can be transformed to viewing coordinates. The necessity for view coordinates is that certain operations, such as clipping, are more conveniently implemented in the view space.

At a minimum, a viewing system must have the following features. First a viewing system must have a vector which establishes the viewer position and direction within the world

coordinate system. This has already been defined in Figure 2.15 as the view reference point (VRP) and view plane normal (VPN) vector. Second, a view normal plane (VNP) must be defined a distance (d) from the VRP. This is shown in the Figure 2.16 below:

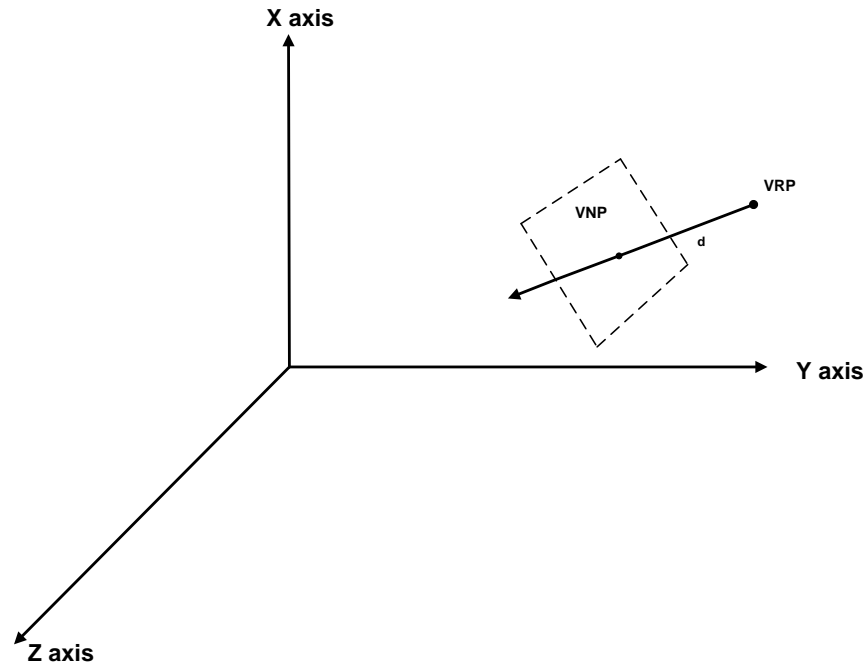


Figure 2.16: Viewing plane

The viewing plane is used to project the 2D image of the 3D scene. Third, using the VRP and VNP and a perpendicular vector to the VPN, the view up vector (VUP), a view coordinate system with a center at the VRP and normal to VNP is defined. This is shown below in Figure 2.17.

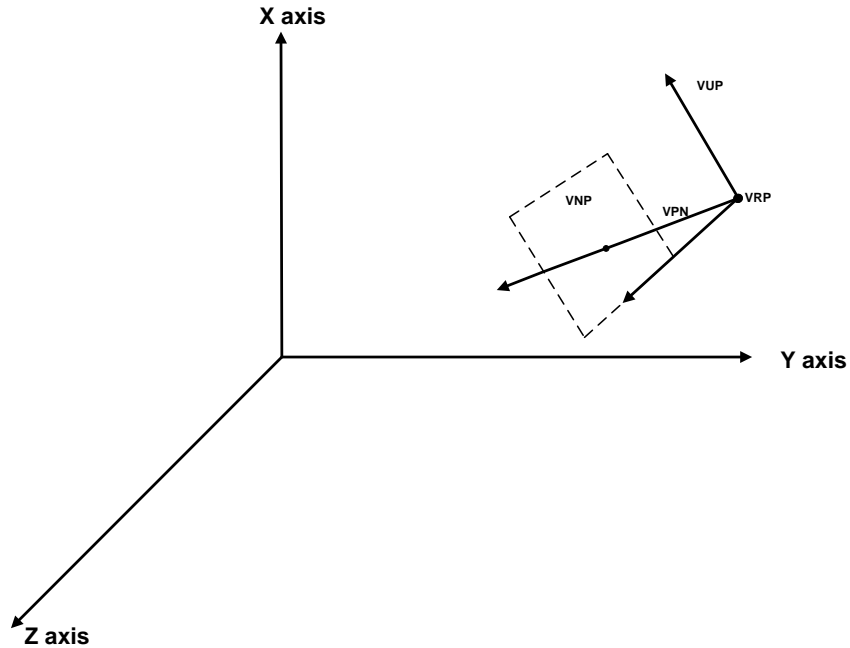


Figure 2.17: View coordinate system.

Lastly, the VRP and VNP combine together to form a view volume. This volume will be used in clipping objects outside the field of view. As will be shown later, the viewing volume has well defined planer intersection equations that makes clipping more convenient and justifies the use of view coordinates. The view volume is shown in Figure 2.18.

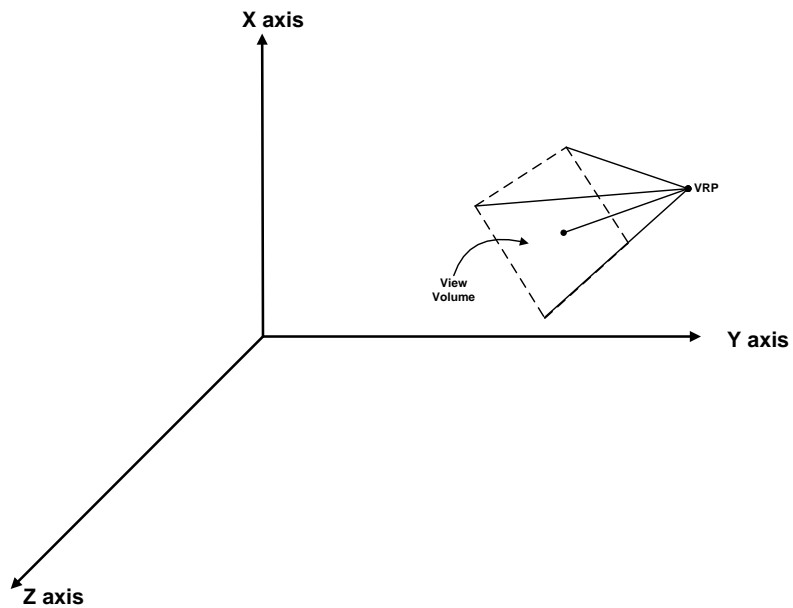


Figure 2.18: View Volume

Using this viewing system, the view reference point can be positioned anywhere in the world coordinate system and pointed in any direction.

Now that the notion of the viewing coordinate system has been explained, the affine geometric transformation used to convert from world coordinate space to view space will be presented. To do the conversion, a composition of two matrices is required. The first transformation matrix is a translation matrix and is used to move the view reference point in the world coordinate system to the origin of the view coordinate system. Second a rotational transformation matrix is used to align the view plane normal with the z axis of the view coordinate system. The composed transformation matrix is shown below:

$$M = R(\theta_z, \theta_x, \theta_y) \cdot T(-c_x, -c_y, -c_z)$$

$$M = \begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & r_{12} & 0 \\ r_{20} & r_{21} & r_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} r_{00} & r_{01} & r_{02} & -c_x r_{00} - c_y r_{01} - c_z r_{02} \\ r_{10} & r_{11} & r_{12} & -c_x r_{10} - c_y r_{11} - c_z r_{12} \\ r_{20} & r_{21} & r_{22} & -c_x r_{20} - c_y r_{21} - c_z r_{22} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 2.12: View Coordinate Transformation

Note that the rotational matrix $R(\theta_z, \theta_x, \theta_y)$ elements are substitutions for the mixed cosine/sine functions in Equation 2.10. This matrix can be used to move objects to the new view coordinate system defined by axes (U,V,N) shown in Figure 2.19. The figure shows this new coordinate system with respect to the world coordinate system before the transformation.

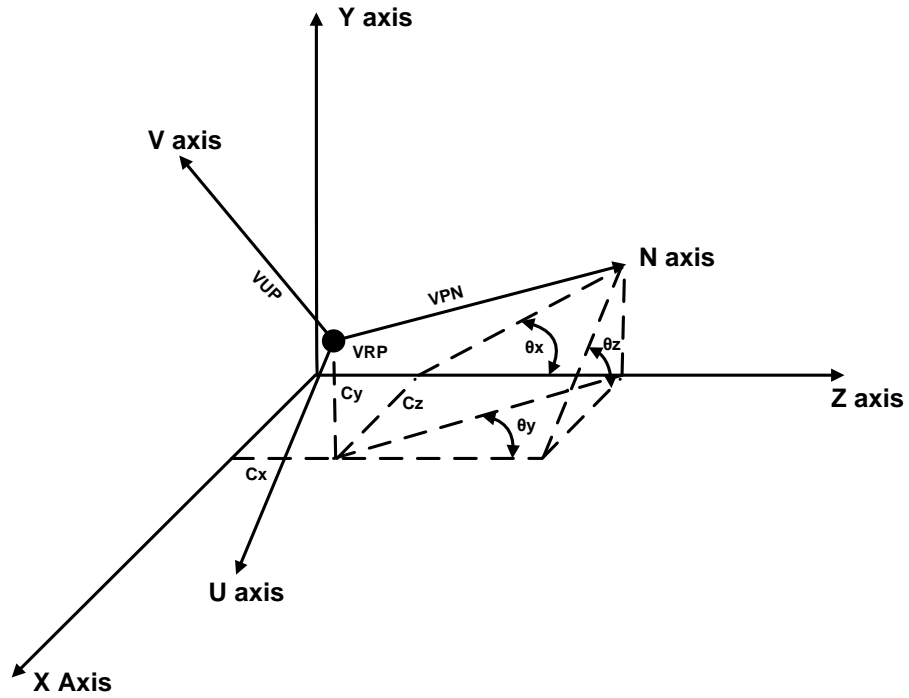


Figure 2.19: Viewing coordinate system defined with World coordinate system.

The result of the transformation has the VRP at the origin, the VPN going straight down the Z axis and the VUP perpendicular to the VPN. The resulting view coordinate system (U,V,N) is shown below:

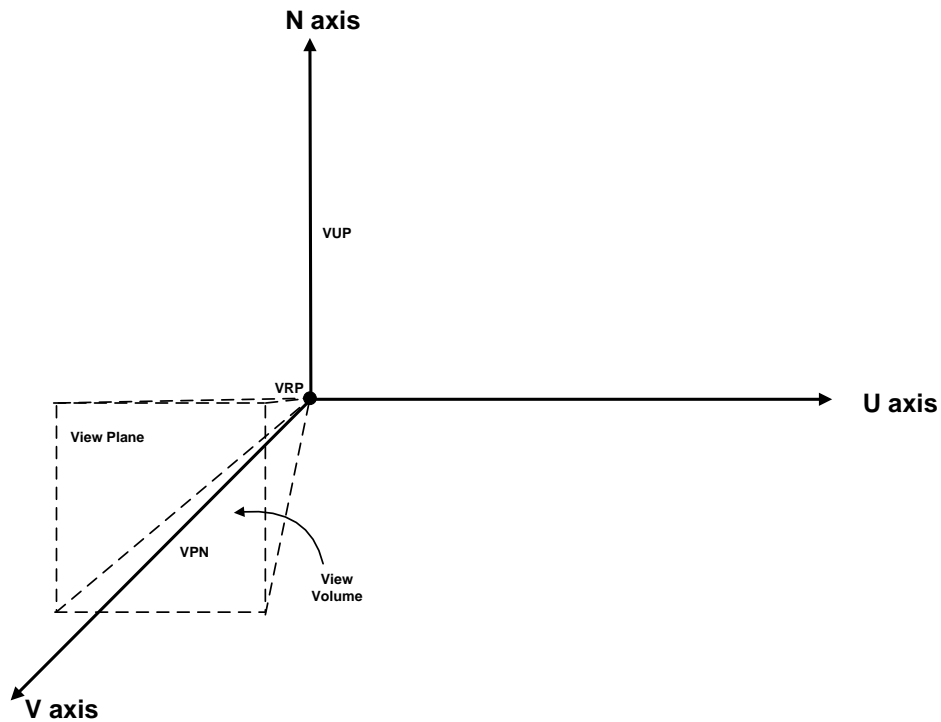


Figure 2.20: Viewing Coordinate System

Now that the viewing coordinate system and view volume have been defined, the scene can be clipped against the view volume and projected onto the view plane.

Projection takes the 3D view and projects it onto a 2D projection plane, much like a movie projector and a projection screen. There are two forms of projection, parallel and perspective. Both projection types have a center of projection (COP), but the projection type is determined by whether the projectors are parallel or join together to form a single point. Both parallel and perspective projections are detailed below.

2.2.4 3D Projections and the Clipping Transformation

The next coordinate space in the graphics pipeline shown in **Error! Reference source not found.** is called the clipping coordinate space. It is called this because this is the space where all clipping against the view volume will take place. This space exists because it both

simplifies the clipping process and projects a 3D image to a 2D plane. The clipping process is discussed in the next sub-section while 3D projections are discussed here.

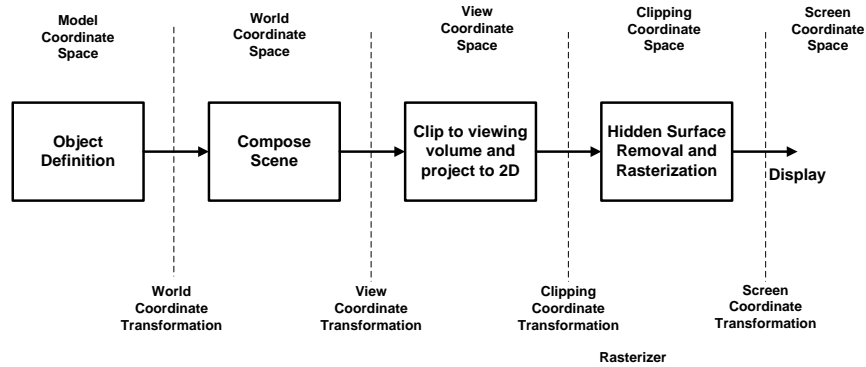


Figure 2.21: Example Graphics Pipeline(6)

The complexity of viewing 3D objects comes from the fact that the objects are described in three dimensions while a computer screen is only two dimensions. To deal with this mismatch between the three dimensional world and the two dimensional screen, the concept of projections are used. A projection is the process of reducing the number of dimensions for a given geometric object. This section details what projections are as well as how to project a 3D environment to a 2D screen.

The types of projections that are dealt with here are planar geometric projections meaning that the objects are projected onto a flat plane as opposed to a curved surface. By projecting onto a flat plane, the equations needed to perform the projection can be greatly simplified. There are two types of planar geometric projections dealt with here, parallel and perspective. Parallel projections have a center of projection (COP) with a distance of infinity from the projection plane known as the far clipping plane. In parallel projection, the projectors (lines that intersect

with the COP) are all parallel with each other, hence the name parallel projection. This forms a cubic viewing volume that is shown in the Figure 2.22.

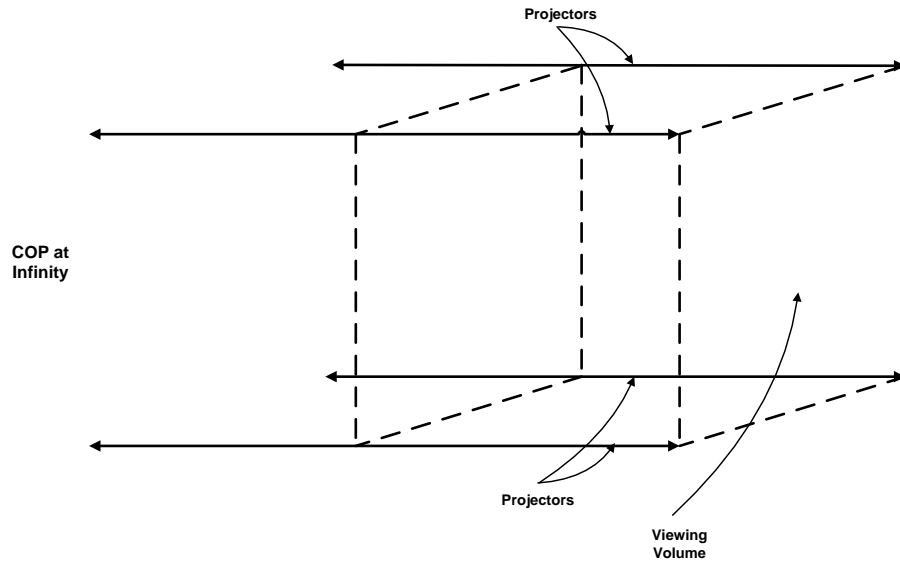


Figure 2.22: Parallel Projection

Perspective in contrast has a finite distance between the center of projection and the projection plane. All of the projectors in perspective projection intersect at the COP and extend outward to the projection plane. This forms a cut off pyramid viewing volume which is shown below.

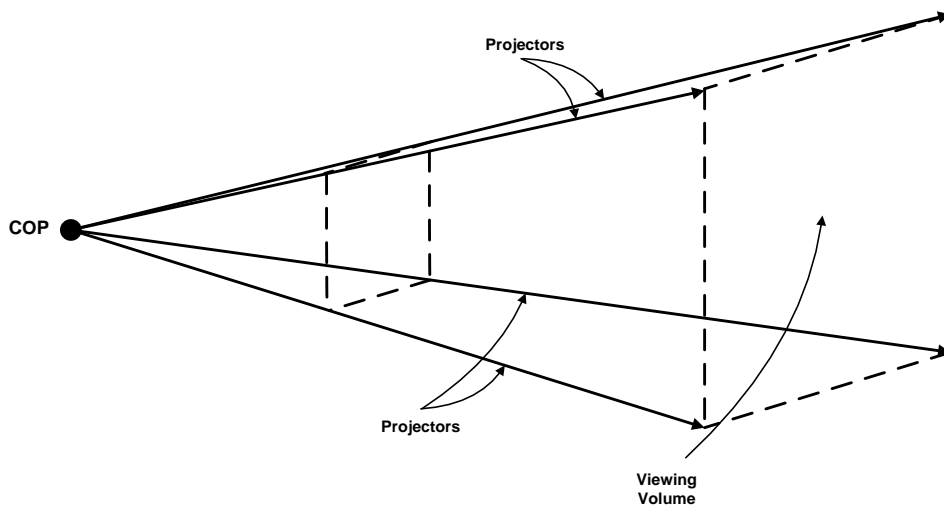


Figure 2.23: Perspective Projection

The difference between these projection types as well as what they are used for is discussed below. This sub-section goes into more detail on the projection types and their differences and presents the transformation matrices necessary to convert from view coordinates to clipping coordinates for each projection.

2.2.4.1 Perspective Projection

Perspective projections are similar to that of how the human eye perceives the world. This effect is known as perspective foreshortening whereby the size of an object varies inversely with distance that the object is from the center of projection. These projections while useful for making realistic looking scenes are not useful for engineering applications which require measuring the size and length of object in 3D (this is where parallel projections come into play).

Figure 2.24 below shows a single point perspective projection from two different views.

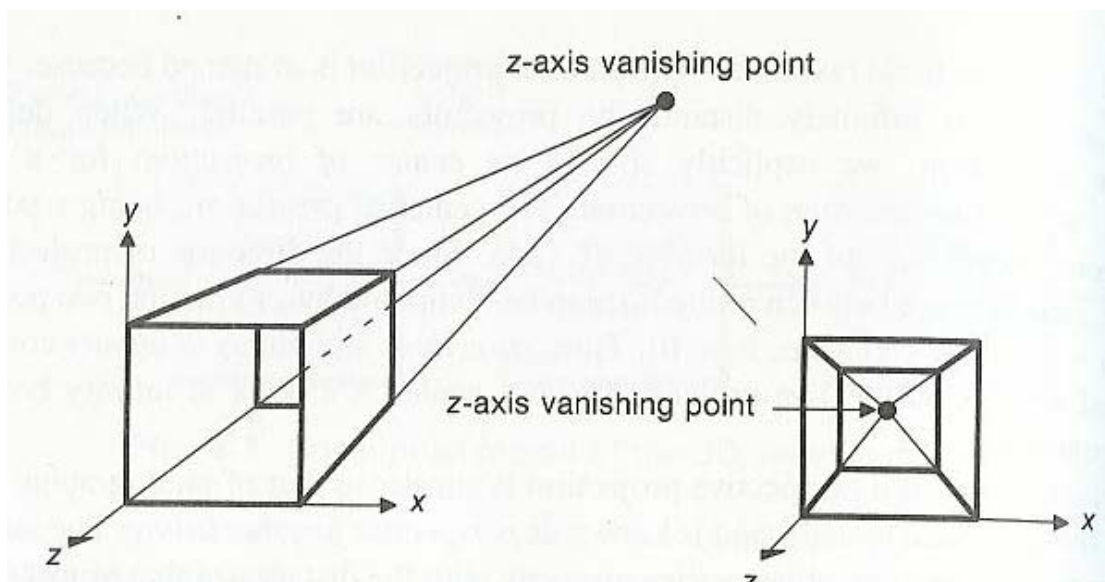


Figure 2.24: Perspective Projection (7)

In a perspective projection, a set of parallel lines, such as the ones in the cube shown above, converge at a point in infinity. This point is known as the vanishing point which just so happens to be the same point as the center of projection (COP). In general, parallel lines in perspective projections only meet at infinity. If the parallel lines are parallel to the z-axis this vanishing point is known as the principle vanishing point. The figure above has a single principle vanishing point, and hence is known as a single point perspective projection.

Figure 2.25 shows how a perspective projection point is derived. $P(x,y,z)$ defined in the view coordinate system is to be projected onto the projection plane a distance d from the center of projection which is normal to the z axis. Point P_p is the projection of this point on the projection plane. The projected point P_p is a two dimensional vector projection plane. The mathematics for perspective single point planar geometric projections can be made fairly simple by making a few assumptions. First it is assumed that the projection plane is located at distance defined by d shown in Figure 2.25. It is also assumed that the center of projection is located at the origin of the viewing coordinate system. In the figure below the center of projection is at the origin of the coordinate system and the projection plane is parallel with the z axis.

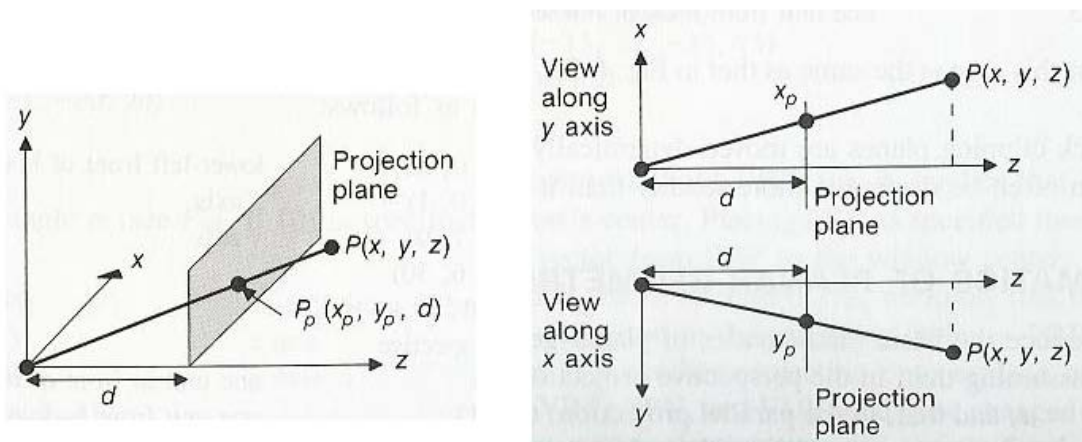


Figure 2.25: Perspective Projection (7)

The object of perspective projections is to find the point P_p , which is the point where point P intersects the projection plane. In order to find this point, the ratio of similar triangles can be used. These ratios and the solutions for x_p and y_p are shown below.

$$\frac{x_p}{d} = \frac{x}{z} \qquad \frac{y_p}{d} = \frac{y}{z}$$

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d} \qquad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}$$

Equation 2.13: Similar Triangle Ratios with solutions for x and y (7).

Just think of the distance d as a scale factor while the z axis causes a closer object to appear larger and further object to appear smaller. The similar triangle equations in Equation 2.13 can be expressed as a single transformation matrix:

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

Equation 2.14: Perspective Projection Transformation Matrix (1)

2.2.4.2 Parallel Projection

Parallel projections in contrast to perspective projections do not mimic the human visual system. Object's sizes in a parallel projection do not vary with depth but instead stay constant. Parallel projections are useful for engineering applications which require measuring the size and length of object in 3D.

Figure 2.26 displays an example of a parallel projection.

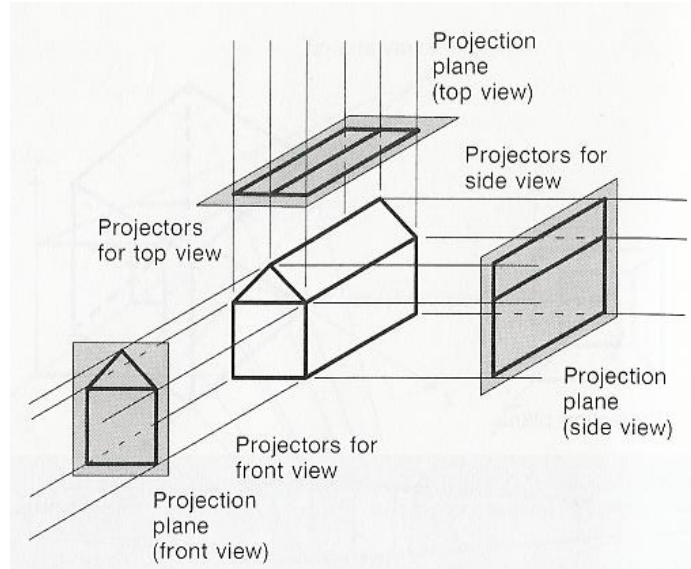


Figure 2.26: Parallel Projection (8)

It can be seen in Figure 2.26 that all the relative lengths of all the sides of the house are preserved. This type of projection is very useful for cad tools and engineering schematics where relative lengths and widths are important.

The parallel projection transformation matrix is fairly trivial. Since the center of projection is at infinity, the distance from the projection plane is also infinity yielding the matrix below.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 2.15: Perspective Projection Transformation Matrix (1)

This matrix simply throws out all the z components in an object.

Using these projection techniques, an object can be converted from 3D coordinates to 2D coordinates. Before projection is done, the pipeline clips the objects to the view volume. From

here the clipping coordinates can be mapped to the screen coordinates in a process known as rasterization. An explanation of clipping is given in the next section.

2.2.5 Clipping

Clipping is the process of removing the parts of a scene that are outside the viewing volume defined in Section 2.2.4. Clipping is important because it filters all non-viewable objects from the graphics pipeline thus increasing performance of the overall system. Many clipping algorithms exist to perform these operations. Since we are dealing with wireframes only, the algorithm dealt with here is the Cohen-Sutherland line clipping algorithm (8). Cohen-Sutherland clips a given line to a rectangular window in 2D or to a cubic or conic volume in 3D.

The key to this algorithm is the initial tests that are performed on each line. First, endpoints are checked for trivial acceptance (the line lies completely in the clipping window or volume). If the object cannot be trivially accepted the object is checked if it can be trivially rejected (the line lies completely outside the clipping window). If a line segment can neither be trivially accepted or rejected then it must be divided into two lines at the edge of the clipping window or volume. These new lines are then checked for trivial acceptance or rejections. If the new lines are not either trivially accepted or rejected the process continues until one line is trivially accepted and the other is trivially rejected. The following sections give a detailed explanation of Cohen-Sutherland clipping in both 2D and 3D.

2.2.5.1 Cohen-Sutherland Two Dimensional Clipping

In a 2D graphics pipeline the Cohen-Sutherland clips a line against a rectangular window. The algorithm divides a 2D space into 9 separate regions. These regions are used to determine if

a line can be trivially accepted, trivially rejected or if the line needs clipped. The regions are shown in Figure 2.27 .

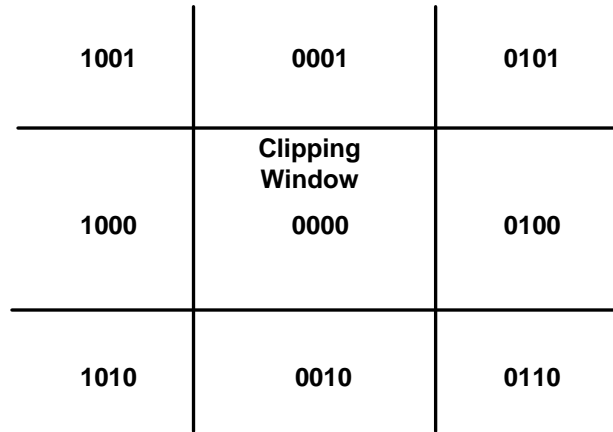


Figure 2.27: Clipping Region Definitions.

From the figure above it can be seen that the nine regions can be defined as a four bit outcode. A four bit outcode is defined for each point of a given line. The outcode bit assignment is shown in the table below.

Table 2.1: 2D Outcode Assignment Table

| Bit Number | Location of End Point | Conditional |
|------------|--------------------------|---|
| First Bit | Above Clipping Window | if $y > y_{max}$ then set bit to 1 else 0 |
| Second Bit | Below Clipping Window | if $y < y_{min}$ then set bit to 1 else 0 |
| Third Bit | Right of Clipping Window | if $x > x_{max}$ then set bit to 1 else 0 |
| Fourth Bit | Left of Clipping Window | if $x < x_{min}$ then set bit to 1 else 0 |

For example, given a general line defined by $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$ the algorithm can be defined in these steps.

1. Compute the outcodes called OC_0 and OC_1 for both points p_0 and p_1 .
2. If OC_0 bitwise OR $OC_1 = 0000$ then the line is trivially accepted and the line is passed to the next stage of the pipeline.
3. If OC_0 bitwise AND $OC_1 \neq 0000$ then the line is trivially rejected and the line is dropped and not passed to the next stage of the pipeline.

4. Otherwise, then the line's non visible portion must be clipped.

Step 4 requires the intersection points with the clipping window to be calculated. Parametric equations can be used to calculate the intersection of a line by introducing a new dimension (t). These new parametric line equations can be used to determine the intersection points with each side of the clipping window. The equations are shown below.

$$x = x_0 + t(x_1 - x_0)$$

$$y = y_0 + t(y_1 - y_0)$$

Equation 2.16: 2D Parametric Equations

Solving for t on the extreme of each clipping window yields four intersection equations. Below is a table which shows how to calculate the new x and y coordinates for a clipped line based on the edge intersected.

Table 2.2: 2D Clipping Intersection Equations

| Clip Edge | Solve for t | Edge intersection equations. |
|----------------|--|---|
| $y = y_{\max}$ | $t = \frac{(y_{\max} - y_0)}{y_1 - y_0}$ | $x = x_0 + \frac{(x_1 - x_0)(y_{\max} - y_0)}{y_1 - y_0}$ |
| $y = y_{\min}$ | $t = \frac{(y_{\min} - y_0)}{y_1 - y_0}$ | $x = x_0 + \frac{(x_1 - x_0)(y_{\min} - y_0)}{y_1 - y_0}$ |
| $x = x_{\max}$ | $t = \frac{(x_{\max} - x_0)}{x_1 - x_0}$ | $y = y_0 + \frac{(y_1 - y_0)(x_{\max} - x_0)}{x_1 - x_0}$ |
| $x = x_{\min}$ | $t = \frac{(x_{\min} - x_0)}{x_1 - x_0}$ | $y = y_0 + \frac{(y_1 - y_0)(x_{\min} - x_0)}{x_1 - x_0}$ |

Based on the bits set in the outcode the appropriate edge intersection equation is used to acquire the new x and y coordinates where the line intersects the clipping edge. These new points are then put through steps 1 through 4 once again. If the line is trivially accepted clipping is completed, otherwise steps 1 through 4 are repeated until trivial acceptance occurs.

An example of clipping can be seen in the Figure 44 below. With line KL both points K and L have an outcode of 0000 because both points are in the clipping window. This line will be trivially accepted. With line MN point M has an outcode of 0110 and N has an outcode of 0100. The bitwise AND of M and N's outcode yields 0100 which is not zero and hence the line should be rejected.

For the first non-trivial cases AB needs to be clipped. Looking at the figure, the resulting outcode of line AB will be 0000 for A and 1001 for B. The logical OR of the outcodes of A and B are not zero and the logic and is zero therefore the line can be neither trivially accepted nor trivially rejected. B's outcode of 1001 indicates that it is above and to the left of the clipping window. Since B's outcode has two bits set either the y_{max} intersection equation or the x_{min} intersection equation from Table 2.1 must be used to calculate the intersection. If y_{max} 's calculation is done first the new line will be AD and the new resulting outcode for D will be 0000. Since A and D have an outcode of 0000 the line can be trivially accepted. On the other hand if x_{min} 's intersection equation is calculated first then the new line will be AC and C's new resulting outcode will be 0001. AC bitwise OR of their outcodes yields a non-zero value of 0001 and hence another iterative step of clipping must occur. Once again the y_{max} calculation is done on AC yielding the new line AD. This is the same line as when the y_{max} edge intersection calculation is done first. Note that order of the edge calculation has no effect except for the fact that if the wrong one is selected then additional steps must be taken.

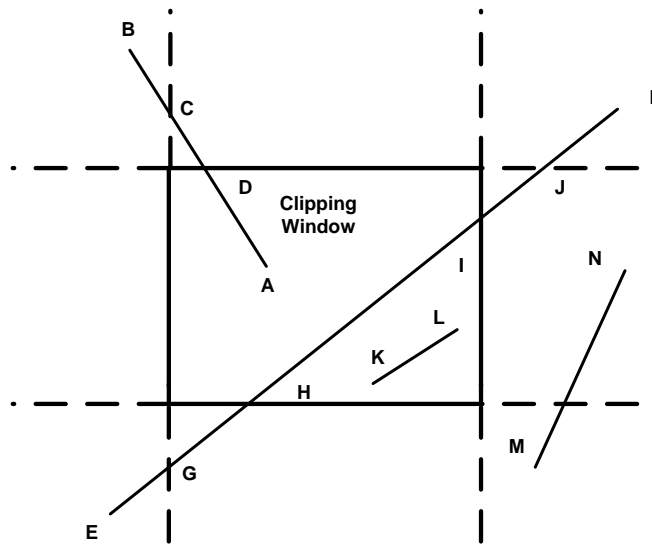


Figure 2.28: Illustration of 2D Cohen Sutherland Clipping.

Line EF is the most complicated case in which both endpoints lie at diagonal corners of the clipping window. In the worst case all four edge calculations may need to be done where as in the best case two edge calculations need to be done.

2.2.5.2 Cohen-Sutherland Three Dimensional Clipping

Cohen-Sutherland clipping can be easily extended to 3D. Instead of a clipping window, in 3D a clipping volume is used. 3D clipping is different based on the projection type. In the case of parallel projection, the clipping volume is a 3D cubic.

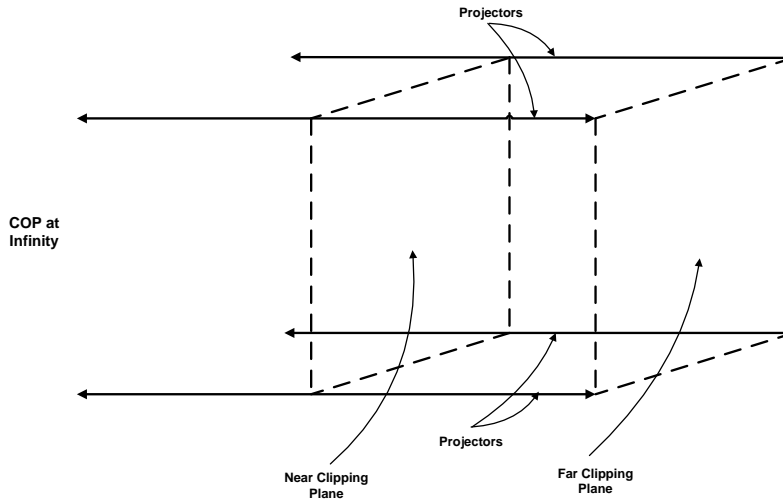


Figure 2.29: Parallel Projection

In the case of perspective projection, the clipping volume is a 3D pyramid.

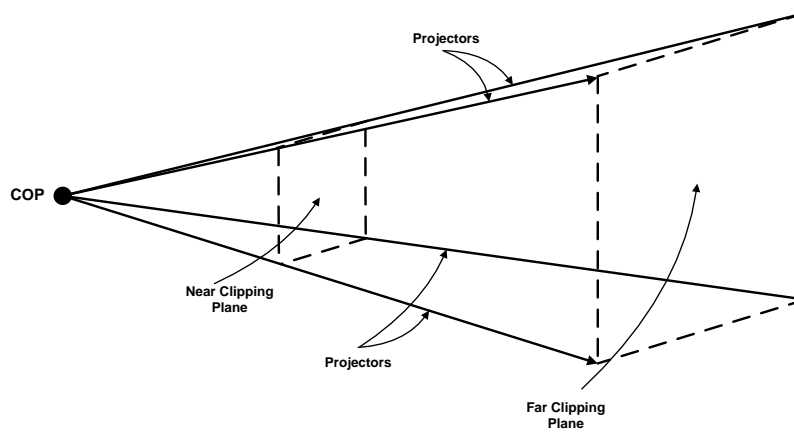


Figure 2.30: Perspective Projection

Parallel projection clipping uses a six bit outcode with a unit clipping cube. The parameters are shown below in Table 2.2.

Table 2.3: 3D Parallel Projection Outcode Assignment

| Bit Number | Location of End Point | Conditional |
|------------|---------------------------------|--------------------------------------|
| First Bit | Above the Clipping Volume | if $y > 1$ then set bit to 1 else 0 |
| Second Bit | Below the Clipping Volume | if $y < -1$ then set bit to 1 else 0 |
| Third Bit | Right of Clipping Volume | if $x > 1$ then set bit to 1 else 0 |
| Fourth Bit | Left of Clipping Volume | if $x < -1$ then set bit to 1 else 0 |
| Fifth Bit | Behind the Clipping Volume | if $z < -1$ then set bit to 1 else 0 |
| Sixth Bit | In Front of the Clipping Volume | if $z > 0$ then set bit to 1 else 0 |

Perspective projection has an six bit outcode which varies with the depth within the conical view volume, hence the z values within the conditionals.

Table 2.4: 3D Perspective Projection Outcode Assignment

| Bit Number | Location of End Point | Conditional |
|------------|---------------------------------|--|
| First Bit | Above the Clipping Volume | if $y > -z$ then set bit to 1 else 0 |
| Second Bit | Below the Clipping Volume | if $y < z$ then set bit to 1 else 0 |
| Third Bit | Right of Clipping Volume | if $x > -z$ then set bit to 1 else 0 |
| Fourth Bit | Left of Clipping Volume | if $x < z$ then set bit to 1 else 0 |
| Fifth Bit | Behind the Clipping Volume | if $z < -1$ then set bit to 1 else 0 |
| Sixth Bit | In Front of the Clipping Volume | if $z > z_{\min}$ then set bit to 1 else 0 |

A 2D line is trivially accepted if both endpoints have an outcode of all zeros and trivially rejected if the bit by bit logical AND of both points does not yield zero. 3D is no different except the clipping volume is defined by six planes as opposed to four edges and 27 unique sections exist as opposed to nine.

The intersection calculations for each of the six sides of the viewing volume can be found once again using parametric equations. Assuming a line from $P_0(x_0, y_0, z_0)$ to $P_1(x_1, y_1, z_1)$ the parametric equations is described as such:

$$\begin{aligned}
 x &= x_0 + t(x_1 - x_0) \\
 y &= y_0 + t(y_1 - y_0) \\
 z &= z_0 + t(z_1 - z_0)
 \end{aligned}$$

Equation 2.17: 3D Parametric Equations

Solving for t on the extremes of the clipping volume yields six planar intersection equations.

Below is a table which shows how to calculate the new x, y and z coordinates for a clipped line based on the plane intersected in both parallel and perspective projections.

Table 2.5: 3D Parallel Projection Clipping Intersection Equations.

| Clip Edge | Solve for t | Planar intersection equations. |
|-----------|------------------------------------|---|
| y = 1 | $t = \frac{(1 - y_0)}{y_1 - y_0}$ | $x = x_0 + \frac{(x_1 - x_0)(1 - y_0)}{y_1 - y_0}$ $z = z_0 + \frac{(z_1 - z_0)(1 - y_0)}{z_1 - z_0}$ |
| y = -1 | $t = \frac{(-1 - y_0)}{y_1 - y_0}$ | $x = x_0 + \frac{(x_1 - x_0)(-1 - y_0)}{y_1 - y_0}$ $z = z_0 + \frac{(z_1 - z_0)(-1 - y_0)}{z_1 - z_0}$ |
| x = 1 | $t = \frac{(1 - x_0)}{x_1 - x_0}$ | $y = y_0 + \frac{(y_1 - y_0)(1 - x_0)}{x_1 - x_0}$ $z = z_0 + \frac{(z_1 - z_0)(1 - x_0)}{x_1 - x_0}$ |
| x = -1 | $t = \frac{(-1 - x_0)}{x_1 - x_0}$ | $y = y_0 + \frac{(y_1 - y_0)(-1 - x_0)}{x_1 - x_0}$ $z = z_0 + \frac{(z_1 - z_0)(-1 - x_0)}{x_1 - x_0}$ |
| z = -1 | $t = \frac{(-1 - z_0)}{z_1 - z_0}$ | $x = x_0 + \frac{(x_1 - x_0)(-1 - z_0)}{z_1 - z_0}$ $y = y_0 + \frac{(y_1 - y_0)(-1 - z_0)}{z_1 - z_0}$ |
| z = 0 | $t = \frac{-z_0}{z_1 - z_0}$ | $x = x_0 + \frac{(x_1 - x_0)(-z_0)}{z_1 - z_0}$ $y = y_0 + \frac{(y_1 - y_0)(-z_0)}{z_1 - z_0}$ |

Table 2.6: 3D Perspective Projection Clipping Intersection Equations.

| Clip Edge | Solve for t | Planar intersection equations. |
|----------------------|--|---|
| y = -z | $t = \frac{(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ | $x = x_0 + \frac{(x_1 - x_0)(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ |
| y = z | $t = \frac{(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ | $x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ |
| x = -z | $t = \frac{(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ | $y = y_0 + \frac{(y_1 - y_0)(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ |
| x = z | $t = \frac{(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ | $y = y_0 + \frac{(y_1 - y_0)(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ |
| z = -1 | $t = \frac{(-1 - z_0)}{z_1 - z_0}$ | $x = x_0 + \frac{(x_1 - x_0)(-1 - z_0)}{z_1 - z_0}$ $y = y_0 + \frac{(y_1 - y_0)(-1 - z_0)}{z_1 - z_0}$ |
| Z = Z _{min} | $t = \frac{(z_{\min} - z_0)}{z_1 - z_0}$ | $x = x_0 + \frac{(x_1 - x_0)(z_{\min} - z_0)}{z_1 - z_0}$ $y = y_0 + \frac{(y_1 - y_0)(z_{\min} - z_0)}{z_1 - z_0}$ |

Once an object is projected and clipped it can then be rasterized onto the screen. This requires transforming the object's endpoints from 2D projected coordinates to display specific screen coordinates. Using these screen coordinates, the rasterizer handles filling in the pixels on the display.

2.2.6 Screen Coordinate Transformation

Screen coordinates are the actual coordinates of the pixels in a particular raster display. The screen coordinate transformation tells the graphics pipeline how to map clipping coordinates onto a raster screen. Generally, this is display dependent and the screen transformation matrix parameters will depend on the display size and type. The figure below shows an example of a square transformed from clipping coordinates to screen coordinates. Note that in Figure 2.31, the unit square in clipping coordinates becomes a rectangle in screen coordinates. This is meant to reflect the fact that most displays have more pixels in the horizontal direction than in the vertical direction.

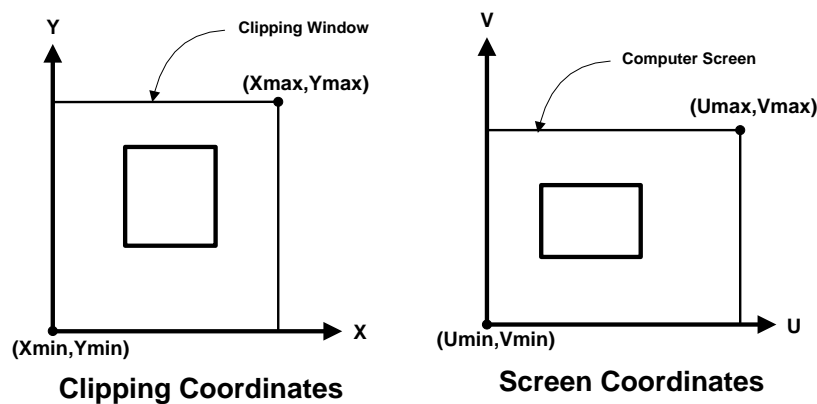


Figure 2.31: Clipping to Screen Coordinates

Where the x,y coordinate system is defined in the clipping space and the u,v coordinate system is defined in screen coordinates. Clipping to screen coordinates requires a scaling transformation

to map the coordinates within the ranges x_{min} to x_{max} and y_{min} to y_{max} to their respective screen coordinates. This scaling transformation is based on the relative ratios of the length and width of the clipping coordinates and the screen coordinates.

In an arbitrary screen translation where the x, y, u and v minimum and maximum values can be any value, the below matrix transformation can be used to transform between clipping and screen coordinates.

$$M_{screen} = T(u_{min}, v_{min}) \cdot S\left(\frac{u_{max} - u_{min}}{x_{max} - x_{min}}, \frac{v_{max} - v_{min}}{y_{max} - y_{min}}\right) \cdot T(-x_{min}, -y_{min})$$

$$M_{screen} = \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & 0 & -x_{min} \cdot \frac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min} \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & 0 & -y_{min} \cdot \frac{v_{max} - v_{min}}{y_{max} - y_{min}} + v_{min} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 2.18: General Screen Transformation Matrix

The two translation matrices $T(-x_{min}, -y_{min})$ and $T(u_{min}, v_{min})$ are used to center the object in each coordinate system. This calculation can be greatly simplified by taking advantage of the fact that screen coordinate minimums can start at zero as well as using the clipping window values discussed in Section 2.2.5. Taking in consideration each of these facts, the following replacements can be made.

$$\begin{aligned}
u_{\min} &= v_{\min} = 0 \\
x_{\min} &= y_{\min} = -1 \\
M_{screen} &= T(u_{\min}, v_{\min}) \cdot S\left(\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}, \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}\right) \cdot T(-x_{\min}, -y_{\min}) \\
M_{screen} &= T(0,0) \cdot S\left(\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}, \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}\right) \cdot T(1,1) \\
M_{screen} &= \begin{bmatrix} \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} & 0 & 0 & -\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} \\ 0 & \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} & 0 & -\frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

Equation 2.19: General Screen Transformation Matrix

Note, scaling from clipping to screen coordinates can result in non uniform scaling factors which result in a distorted object. This can be seen in Figure 2.31 where the cube object after scaling looks more like a rectangle. This is shown because most computer display has a wide aspect ratio where the width of the screen is greater than its height. This issue can be addressed in the projection transformation where scaling factors can be added to the transformation matrix to counteract the phenomena.

Once the objects are converted to screen coordinates, given that a raster display is quantized, the decimal portions of the object's points can be either truncated or rounded. This effectively converts all floating point numbers to an integer representation which can be taken advantage of in Bresenham's line algorithm discussed in the next section.

2.2.7 Rasterization

Rasterization is the process of converting graphics objects to pixels that lie on a 2D grid. Since this design deals only with wireframes, line rasterization is all that is necessary to

implement. The most common line rasterization algorithm is Bresenham's Line Algorithm (9). Bresenham's Algorithm provides a close approximation between the actual line and what is displayed on the screen. This algorithm is efficient using only integer addition and bit shifting.

Since real numbers are used to represent objects and there are only a discrete number of pixels in a display, there will be small errors in representation of a line. This is known as aliasing, and can be reduced with anti-aliasing filters. Anti-aliasing filters, while not in the requirements of this design, are smoothing functions which convolve the surrounding pixels of any given pixel to interpolate what the intensity of each pixel should be.

Anti-aliasing aside, the Bresenham's line algorithm assumes two independent end pixels $P_0(x_0, y_0)$ and $P_1(x_1, y_1)$. An example of a line such as this is shown in Figure 2.32 below.

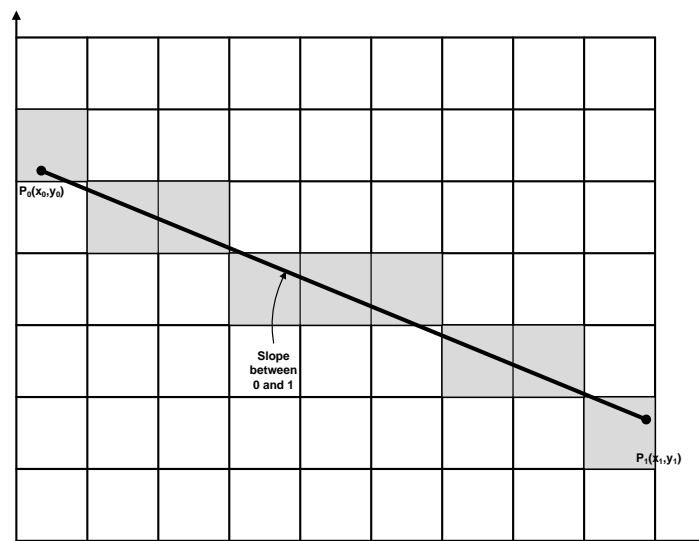


Figure 2.32: Rasterized Line(10)

The equation of the line above is:

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0} (x - x_0)$$

Equation 2.20: General line equation through two endpoints.

In Bresenham's algorithm, the x component is incremented for every raster row. Knowing this fact we can solve for y by adding y_0 to both sides.

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

Equation 2.21: Line equation solved for y

Bresenham's algorithm calculates y as the x component scrolls across the line. Using the y calculated the closest y is chosen.

With the basic Bresenham algorithm, the problem is limited in two ways. First, the slope of the line (M) is restricted to a negative slope between zero and one. Making this assumption, the line drawing routine always increments x as it plots from the starting point to the end point.

Figure 2.33 below shows a line plotted given these assumptions.

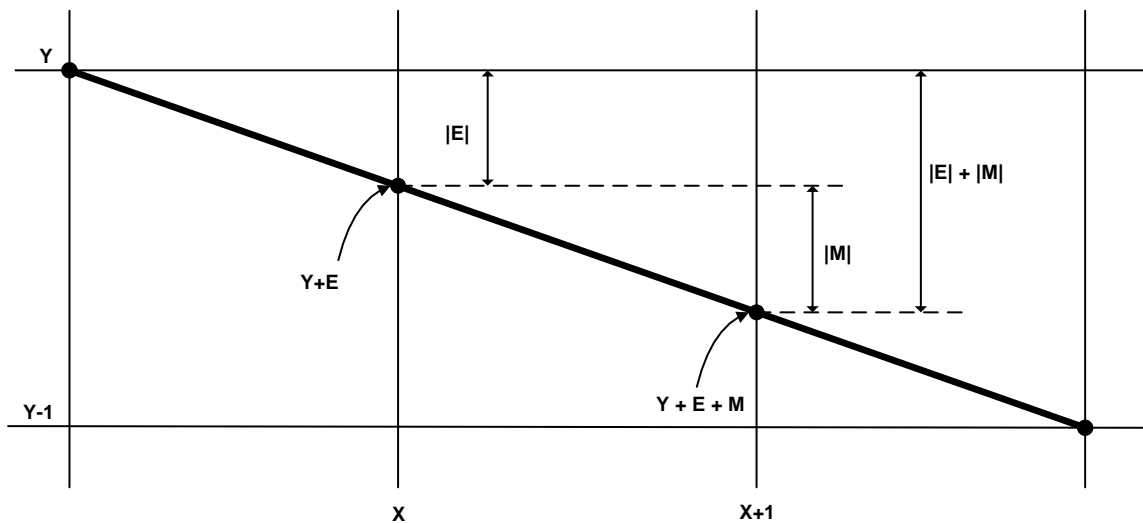


Figure 2.33: Bresenham's Line Algorithm Diagram(11)

Limiting the algorithm in this way leaves the algorithm with a limited number of options as to where to put the next pixel as it traverses the x axis. The choices for the next pixel are either (x+1,y) or (x+1,y-1). At this point the algorithm decides which pixel should come next. Moving from x to x+1 the y coordinate is increased by an amount equal to the slope of the line

(M). Since the real value of y and the quantized value of y in the raster display will usually not be equal. In this case the error (E) is associated with each y coordinate where the real y coordinate is $y+E$. This error lies between $-\frac{1}{2}$ and $\frac{1}{2}$. The next point should be the one with the smallest error. If the current $y + \frac{1}{2} > -(y+E+M)$ then the error is less than the half way point between the current y coordinate and the next coordinate $y-1$. In this case, select $(x+1,y)$ as the next pixel.. Otherwise, the error is big enough that the next y is below the half way point between the current y and the next coordinate $y-1$ so select $(x+1,y-1)$. After each point is calculated, the error is updated depending on which point was plotted. If $(x+1,y)$ was plotted then the error is only increased by the slope (E+M) else if $(x+1,y-1)$ is plotted then the error is increased by the slope plus one (E+M+1). The pseudo code below shows what is described below:

```

E=0; y=y0;
dx=x1- x0; dy=y1- y0;
YStep = -1
M=dy/dx;
For x from x0 to x1 loop
  Plot(x, y); // Plots a point on the raster display.
  if (E + M<0.5) then
    E = E + M;
  else
    y=y+Ystep;
    E=E+M- Ystep;
  end if;
end for;

```

Figure 2.34: Line Rasterization Pseudo code.

This line algorithm works great for lines with a negative slope between zero and one, but what about lines with other slopes. The solution lies in simply changing the order of the end points or swapping the x and y coordinates. By doing this, all lines can be rasterized in the same way. Figure 2.35 shows what coordinates need swapped and what the Y step polarity should be for each octant in the graph.

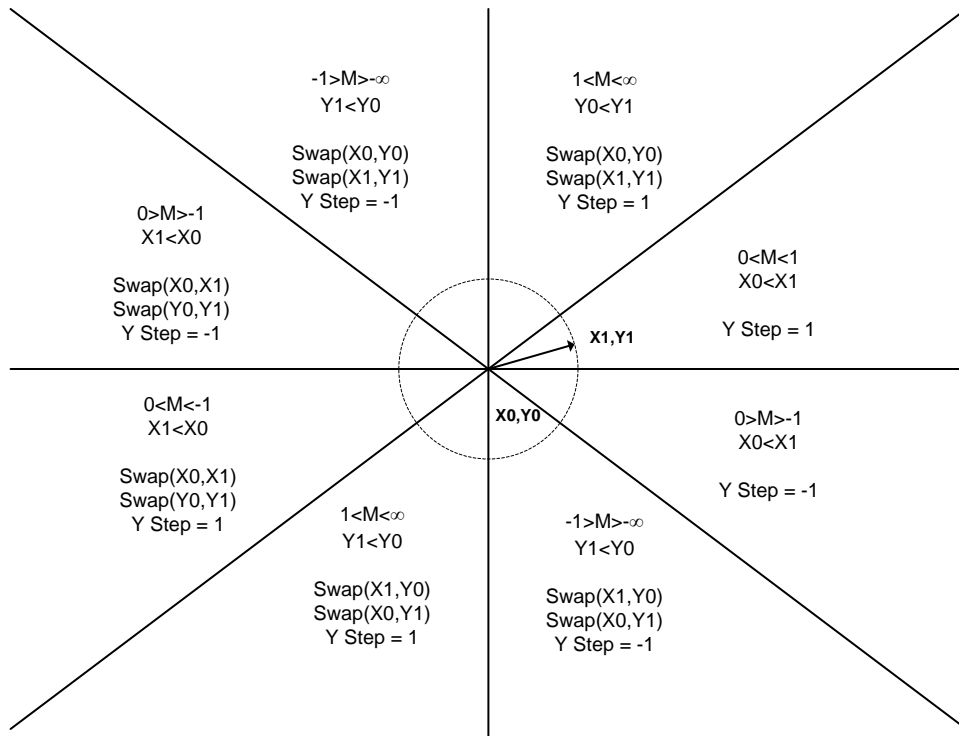


Figure 2.35: Slope Octet Ranges

Using this algorithm the final set of the pipeline is complete. Pixels can now be sent to the computer monitor to be displayed. The next section will use all the mathematical principals, transformations, and graphics operational algorithms to present a FPGA prototype design which meets the requirements set forth in the interdiction.

3.0 THE GRAPHICS PROCESSING UNIT

In Section 2.0 a background on computer graphics was presented. The background information consists of various algorithms used for rendering objects within a virtual scene. These algorithms relate directly to the requirements presented in Section 1.2. Today, most commercial graphics processors are designed using custom designed ASICs. The purpose of this section is to present a GPU design which can be efficiently implemented on FPGA technology.

Before the design is presented, recall the design requirements from Section 1.2 relisted below:

- Graphics processing engine capable of rendering in both 2D and 3D.
- Rendering of wireframe objects.
- Support for a free roaming point of view (or camera) in 3D.
- Support standard display interfaces.

These requirements constitute a fully functional graphics processing unit. Given these requirements and based on the computer graphics overview given in Section 2, a set of design specifications can be derived. The functional specifications to implement the graphics processing requirements are listed here:

- Geometric transformation engine capable of scaling, rotation and translation of objects in a 3D or 2D environment using floating point arithmetic.
- Projection of 3D objects on to a 2D surface.

- Clipping of objects outside the screen's viewing area.
- Wire frame rasterization.
- Standard VGA/DVI display interface.
- Control interface for manipulating graphics objects.

In addition, some general performance constraints to aim for are put in place. The performance specifications are listed here.

- Geometry engine which can process around 8 million polygons per second for 3D and 24 million lines per second in 2D.
- Pixel fill rate of 100 million pixels per second.
- Support a standard VGA resolution of 640x480 at 60 frames per second.
- Support for 262,114 colors (18 bit color).

Note that these performance specs are much lower than today's top of the line commercial offerings. For instance, the Geforce 8800 GS (2) supports resolutions up to 1920x1080 and has a pixel fill rate 26.4 billion pixels per second. Far greater than what is being aimed for here. This is acceptable because this thesis is just testing the feasibility of an FPGA graphics based design, not aiming to compete with today's top of the line graphics processors.

In order to satisfy these specifications an architecture which is feasible to implement in an FPGA must be derived. Recall from Section 2.0 the generic graphics pipeline discussed shown again in Figure 3.2. The pipeline requires design for the transformation matrices to go from coordinate system to coordinate system. In addition, special purpose logic is needed to accelerate both clipping and rasterization. Also, a method for configuring the transformation matrices and storage mechanism for the graphics objects.

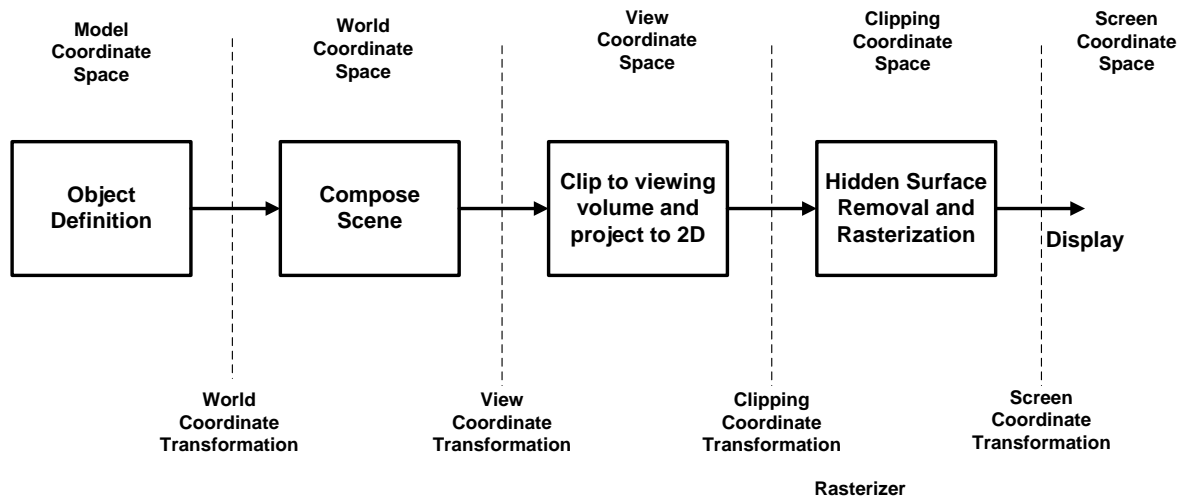


Figure 3.1: Graphics Pipeline(1)

A design which can handle all these operations efficiently needs hardware acceleration of the entire GPU pipeline. This is because the GPU pipeline is where the bulk of the calculations are done. The GPU pipeline requires hardware acceleration for both the matrix transformations as well as Cohen-Sutherland clipping to perform efficiently at 60 frames per second. A hardware accelerated Bresenham's line algorithm is also needed. An implementation of this algorithm in hardware requires frame buffering to be implemented efficiently. This is because raster displays expect the values for each pixel to be driven out according to the display's timing specs. Another fact is that each pixel is only driven out once making pushing each object to the display one at a time impossible. To get around this issue using a frame buffer allows all the objects in the virtual scene to be updated with the final scene with all the rendered objects stored in the buffer. From there the image can be driven out to the display properly. Another issue that needs to be handled is storage of graphics objects as well as how they are sent to the GPU pipeline. A simplistic way to handle this is to use a central control processor to store and manipulate the graphics objects in a scene and when required, drive these objects to the graphics pipeline.

The resulting top level architecture that was developed to meet all these requirements is shown in Figure 3.2.

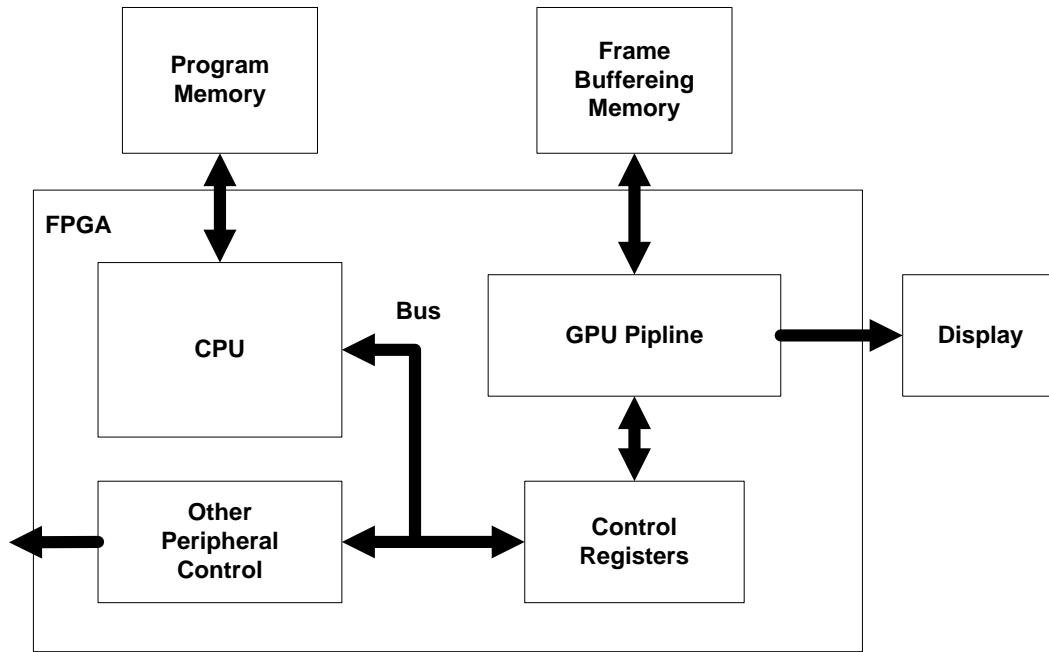


Figure 3.2: GPU Top Level Block Diagram

The GPU in Figure 3.2 needs to be able to process graphics objects as quickly as possible. Though many of these operations could be done using a general purpose CPU using software, the matrix transformations, clipping operations, and wire-frame rasterization discussed in Section 2.0 require many floating point multiplications, additions and divisions. Floating point operations such as these take many cycles in a general purpose CPU. This fact makes implementing these operations in software for real-time graphics processing of complex scenes unrealistic with CPUs. To achieve the performance specifications, hardware acceleration of the matrix transformations, clipping operations and rasterization is necessary. The GPU implements these complex operations using hardware acceleration. The GPU pipeline also handles driving out the images to the display interface at 60 frames per second as required in the specs. Lastly, a

frame buffering memory is required to hold a complete frame of data which is to be driven out to the display. The frame buffer allows alterations to be done to the current frame without effecting what is currently being displayed over the monitor.

The matrix transformation required for graphics rendering necessitate many trigonometric functions in addition to the floating point multiplication, additions and divisions. This is because each transformation matrix has different matrix elements based on the function of the transformation matrix. The CPU in Figure 3.2 aids in the calculation of these matrix elements. Using the CPU for this purpose simplifies the hardware design. Implementing custom logic for each of the different matrix element calculations would require a large amount of custom logic as well as vastly increase the verification effort needed. Using the CPU does not greatly affect GPU performance because the matrix elements for the transformation matrices, as will be shown, only need to be updated either every frame or only once at power up. Using this hybrid CPU/Graphics Pipeline approach provides a simplistic design that performs well and takes up less area on the FPGA.

The CPU requires a method for interfacing to the graphics pipeline. Control registers are used because they can be accessed easily by the processor as memory mapped IO. These control registers are used to update the graphics pipeline's transformation matrices as well as to push new graphics objects onto the pipeline. The CPU also initializes any other peripherals via the memory mapped IO registers.

In this section the high level GPU design above has been presented. The GPU is designed with no particular development platform or FPGA vendor in mind making the design easily portable to other FPGAs. In addition, this section goes in to further details on the how the high level design in Figure 3.2 satisfies the specifications presented above.

3.1 GRAPHICS PIPELINE

Recall the pipeline presented in Section 2.0 and shown in Figure 3.3. This pipeline shows all the necessary steps for converting a list of abstract defined graphics objects to rasterized pixels on a computer screen. This section uses this pipeline as a basis for the FPGA hardware pipeline presented here.

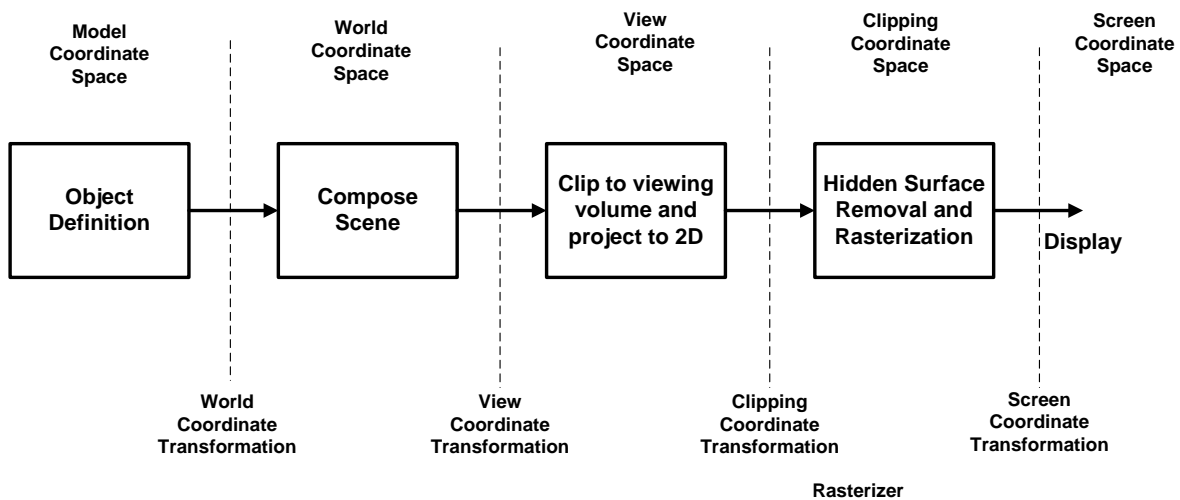


Figure 3.3: Computer Graphic Pipeline(1)

The above pipeline requires many operations to be performed on a graphics object before it is displayed on the screen. In order to render as many graphics objects as possible in the 60 frames per second window specified, a fully pipelined graphics engine was designed. A pipeline maximizes throughput so that object rendering can occur fairly quickly. Also, other techniques like clipping and the use of only wireframe rasterization improve the frame rate and increase the maximum number of objects that can be rendered per frame. This pipelined graphics processing unit is known as the graphics pipeline.

The graphics pipeline is the heart of the graphics processing unit. It takes objects pushed in by the CPU and does world, view, projection and screen coordinate transformations. It also handles clipping to the viewing volume and rasterizing the wireframe representation of the objects defined. Lastly it handles buffering the rasterized data and then pushing out the correct display interface signals in order to drive an external monitor. Figure 3.4 shows a top level diagram of the 3D graphics pipeline.

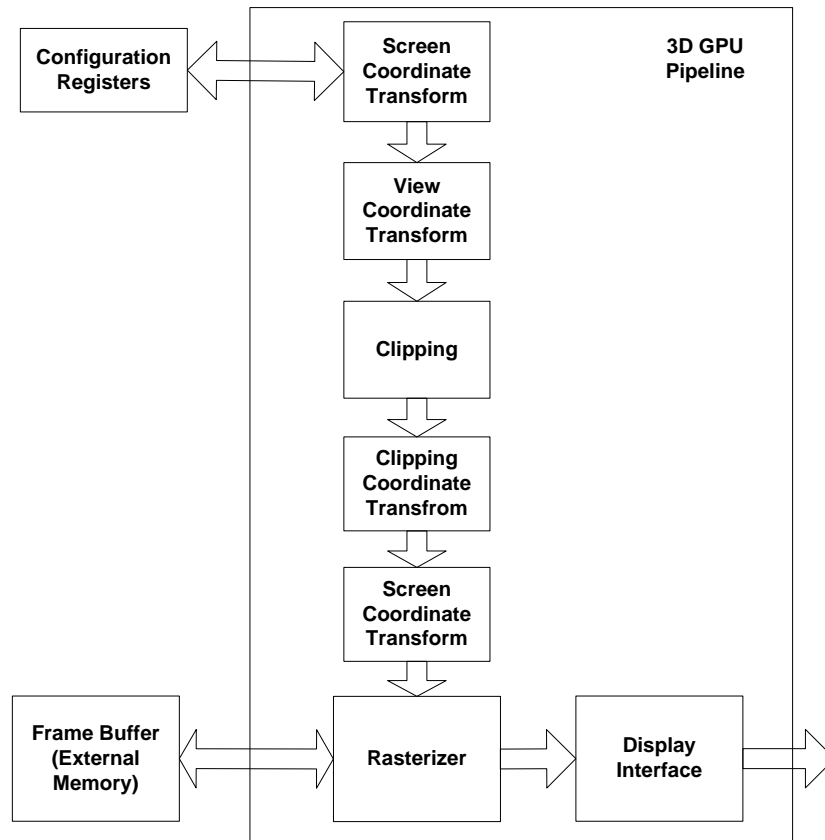


Figure 3.4: 3D Graphics Pipeline

The 2D graphics pipeline is very similar. The only difference is that in 2D there is no notion of view coordinates and projection to 2D is not necessary. This is because a 2D image is already in two-dimensions and can natively be displayed on a 2D monitor. These two blocks are removed and the resulting pipeline is shown in Figure 3.5:

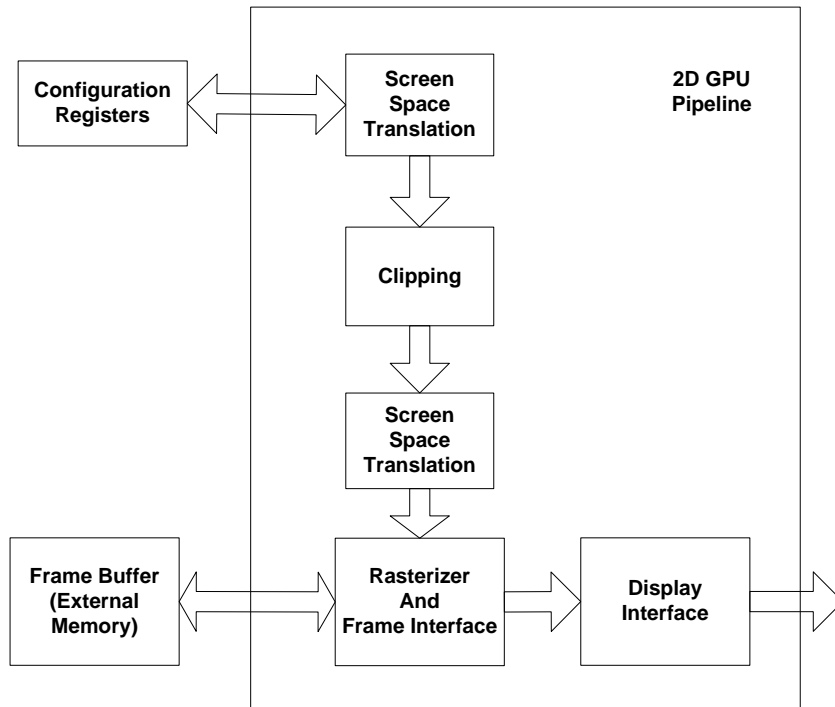


Figure 3.5: 2D Graphics Pipeline

The screen, view, projection and screen translation matrices, can be handled by a common matrix multiplication accelerator block. This is because, the CPU handles calculating the matrix elements for each translation matrix. The elements for each of the four translation matrices are what make each matrix unique. The matrix multiplication accelerator design is presented in Section 3.1.1. The pipeline also handles clipping to the viewing volume using the Cohen-Sutherland's algorithm discussed in Section 2.2.5 and the design is presented in Section 3.1.2. Bresenham's algorithm is then used to rasterize the clipped objects and to push them to the frame buffer. This design is presented in Section 3.1.3. Lastly, Section 3.1.4 discusses how data is pulled from the frame buffer data and pushed to the display interface.

3.1.1 Matrix Multiplier Accelerator

Sections 2.2.2, 2.2.3, 2.2.4, and 2.2.6 overview coordinate transformations that all require the multiplication by a transformation matrix. All of these transformations are required to properly implement a geometry engine discussed in the design specifications. An operation such as matrix multiplication can be a computationally expensive operation for a CPU to handle in software. A 4x4 matrix multiplication requires 16 multiplications and 12 additions. Given these facts and the fact that CPUs execute sequentially, the latency for a single matrix multiplication can be hundreds of CPU cycles. To improve efficiency, matrix multiplication can be implemented in hardware. Fortunately, multiplication of a matrix requires operations between rows and columns of a matrix. These operations are independent and can be implemented in parallel. Parallel operations such as these lend themselves to hardware acceleration.

The matrix multiplier accelerator block is used by the graphics pipeline to translate, scale and rotate objects to different coordinate systems as discussed in Section 2. In 2D, an end point of a line is defined by a homogenous vector which consists of a x value, a y value, and a homogenous value w. Similarly, a 3D end point is also defined by a homogenous vector containing an additional z value to the x, y and w values. Homogenous vectors are discussed in Section 2.1.1. These vectors are multiplied by a transformation matrix which result is a coordinate translation of the vector.

A matrix multiplication is a rather complex operation and if it was to be done all in a single clock cycle, the FPGA resource requirements would be massive. The matrix multiplication is depicted below.

$$\begin{bmatrix} v_x' \\ v_y' \\ v_z' \\ v_w' \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x m_{00} + v_y m_{01} + v_z m_{02} + v_w m_{03} \\ v_x m_{10} + v_y m_{11} + v_z m_{12} + v_w m_{13} \\ v_x m_{20} + v_y m_{21} + v_z m_{22} + v_w m_{23} \\ v_x m_{30} + v_y m_{31} + v_z m_{32} + v_w m_{33} \end{bmatrix}$$

Equation 3.1: Matrix Multiplication(6)

As stated above, a single cycle matrix multiplication requires 16 floating point multiplies and 12 floating point additions. Such an implementation would have huge resource requirements. For these reasons it was decided to use a four step matrix multiplier that utilizes resource sharing. Figure 3.6 illustrates the matrix multiplier design.

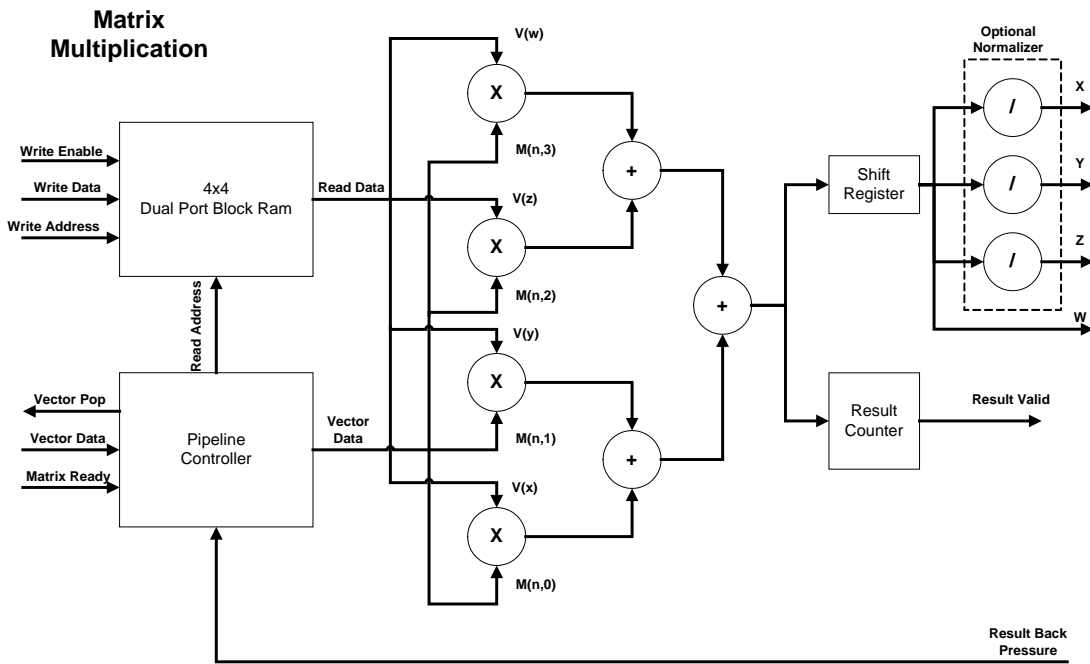


Figure 3.6: Floating Point Matrix Multiplication Block Diagram

Each matrix multiplication block contains a 4x4 dual port block ram. This ram is used to store the contents of the transformation matrix with each address location storing an entire matrix row. An external CPU is responsible for calculating the matrix elements and programming the block ram by setting the write enable signal and applying the matrix data and address to the write

data and write address signals. The processor is also responsible for driving the matrix ready signal when all the transformation matrix entries have been programmed. Once the ready signal is received data can be retrieved by the pipeline controller from an upstream FIFO connected to the vector pop and vector data signals. The pipeline controller is also responsible for controlling selection of the matrix row to read from the dual port block ram. The data read from the RAM is pushed to the floating point multipliers. The multipliers accept the vector data V and the matrix row $M(n)$ and multiply the appropriate component as shown in Equation 3.1. The multiplied data is then pushed through two addition stages effectively adding all the products together. The results of the additions are pushed into a shift register where each resulting component is stored until all four components of the new vector are stored in the shift register. Once all four entries are present in the shift register, result valid pulse is set and the new vector can be read. When other operations such as clipping or rasterization need the raw normalized x , y , and z vector coordinates from a matrix calculation, normalization is needed. The normalization consists of dividing the x , y and z coordinates by w . This returns the vector to homogenous standard form where $w = 1$. The normalizer block is optional and hence is only instantiated when necessary. Another thing to note is that the same matrix multiplier is used for both 2D and 3D, the only difference being that in 2D the z coordinate is always set to zero.

In order to meet the polygons per second performance specification each floating point operation is fully pipelined. Fully pipelining the design allows the logic to be clocked at a high clock speed. Since the multipliers and adders can accept new data every cycle and using the fact that it takes four cycles to execute a matrix multiplication, the number of polygons per second this block can process can be calculated. Assuming a 100MHz (or 10ns period) clock and ignoring the latency of the floating point multiplier, the polygons per second can be calculated by

dividing $100\text{MHz} / (4 \text{ cycles} * 3 \text{ Vectors per polygon})$. $100\text{MHz}/12$ is 8.33 Million polygons per second. Assuming 60 frames per second 138,888 thousand polygons can be processed per frame. This meets our original performance specifications.

3.1.2 Clipping Design

Clipping is necessary to properly render lines that are not completely within the viewing volume defined in Section 2.2.5. In addition, objects completely outside the viewing volume can be dropped from the pipeline improving the GPU's performance. The performance improvement is achieved by reducing the amount of line rasterization needed for partially obscured objects as well as eliminating rasterization completely for invisible ones.

The Cohen-Sutherland line clipping algorithm provides an efficient way of implementing clipping in hardware. As discussed in Section 2.2.7, the Cohen-Sutherland algorithm determines whether a line can be trivially accepted, trivially rejected, or if it must be clipped. While this algorithm could be implemented in software, clipping involves calculating the intersection of a line with the viewing volume. The intersection calculation equations involve floating point multiplications, additions and divisions which require many CPU cycles to execute. As with the matrix multiplier accelerator block, in order to achieve the polygons/lines per second performance specifications requires hardware acceleration is required.

Figure 3.7 presents the top level design developed to implement the Cohen-Sutherland clipping algorithm in hardware. The clipping hardware uses floating point arithmetic to calculate the intersections

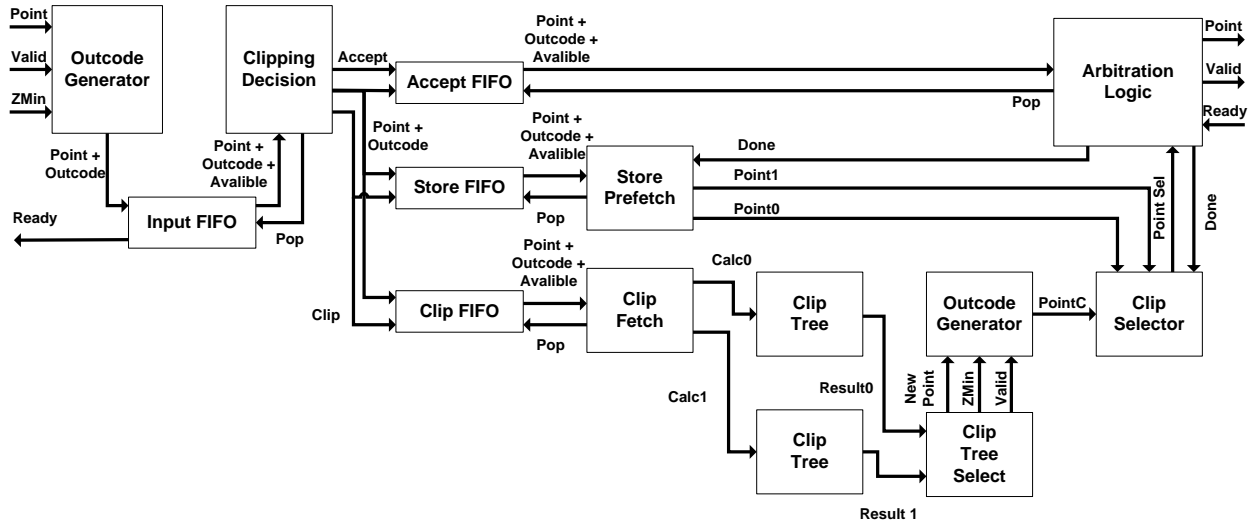


Figure 3.7: Clipping Logic

This design is pipelined in order to provide maximum throughput as well as to not provide a significant bottleneck to the matrix multiplication accelerators. The next few paragraphs give some detail on the functional blocks with in the design.

Recall that Cohen-Sutherland clips lines to either a rectangular window in 2D or a conic volume in perspective 3D. Only perspective projection clipping will be implemented because perspective projection mimics the human visual system unlike parallel projection. This particular clipping algorithm is an iterative procedure where in the initial stage a line is checked to either be completely inside, outside, are partially within the viewing volume. In order to determine where a line lies in respect to the viewing volume, the Cohen-Sutherland uses what is known as an outcode. The outcode is a bit vector which each bit represents which boarder a line violates. For 2D, recall the outcode table from Section 2.2.5.1 shown again in **Table 3.1**.

Table 3.1: 2D Outcode Assignment Table

| Bit Number | Location of End Point | Conditional |
|------------|--------------------------|--|
| First Bit | Above Clipping Window | If $y > y_{\max}$ then set bit to 1 else 0 |
| Second Bit | Below Clipping Window | If $y < y_{\min}$ then set bit to 1 else 0 |
| Third Bit | Right of Clipping Window | If $x > x_{\max}$ then set bit to 1 else 0 |
| Fourth Bit | Left of Clipping Window | If $x < x_{\min}$ then set bit to 1 else 0 |

Similarly for 3D, the outcode is calculated using the table from Section 2.2.5.2.

Table 3.2: 3D Perspective Outcode Assignment Table

| Bit Number | Location of End Point | Conditional |
|------------|---------------------------------|--|
| First Bit | Above the Clipping Volume | if $y > -z$ then set bit to 1 else 0 |
| Second Bit | Below the Clipping Volume | if $y < z$ then set bit to 1 else 0 |
| Third Bit | Right of Clipping Volume | if $x > -z$ then set bit to 1 else 0 |
| Fourth Bit | Left of Clipping Volume | if $x < z$ then set bit to 1 else 0 |
| Fifth Bit | Behind the Clipping Volume | if $z < -1$ then set bit to 1 else 0 |
| Sixth Bit | In Front of the Clipping Volume | if $z > z_{\min}$ then set bit to 1 else 0 |

The first stage of the pipeline is the outcode generator. The outcode generation block in Figure 3.7 involves comparing the endpoints to the border of the view window or view volume. In order for the design to execute these operations efficiently in hardware, floating point comparators are used. The outcode generator design is shown in Figure 3.8.

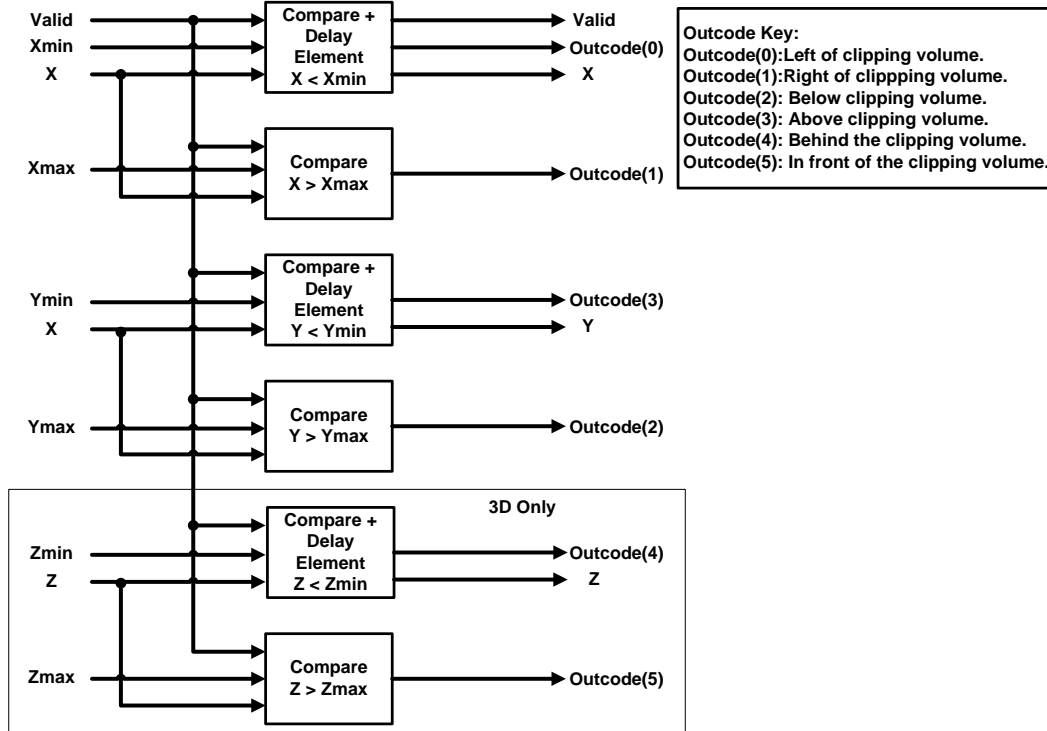


Figure 3.8: Outcode Generator for Clipping Logic.

The outcode has six floating point compares between the line endpoints and the viewing volume's borders (only four in 2D). The compare blocks take the incoming x, y, and z coordinates and compare them against the clipping volume or window. If the point is outside the volume then the corresponding outcode bit is set. Delay elements store the incoming x, y, and z coordinates to be pushed into the input FIFO block along with the outcode.

Clipping requires determining whether a line lies within the viewing volume, outside the viewing volume, or partially within the viewing volume. In order to accomplish this, the outcodes calculated in the outcode generator are compared to each other. The second pipeline stage in the clipping logic pipeline is the clipping decision block. The clipping logic takes the outcodes generated by the outcode generator for both endpoints of a line and compares them. Based on the comparisons, the location of the line with the viewing volume can be established. The state machine in Figure 3.9 represents this process.

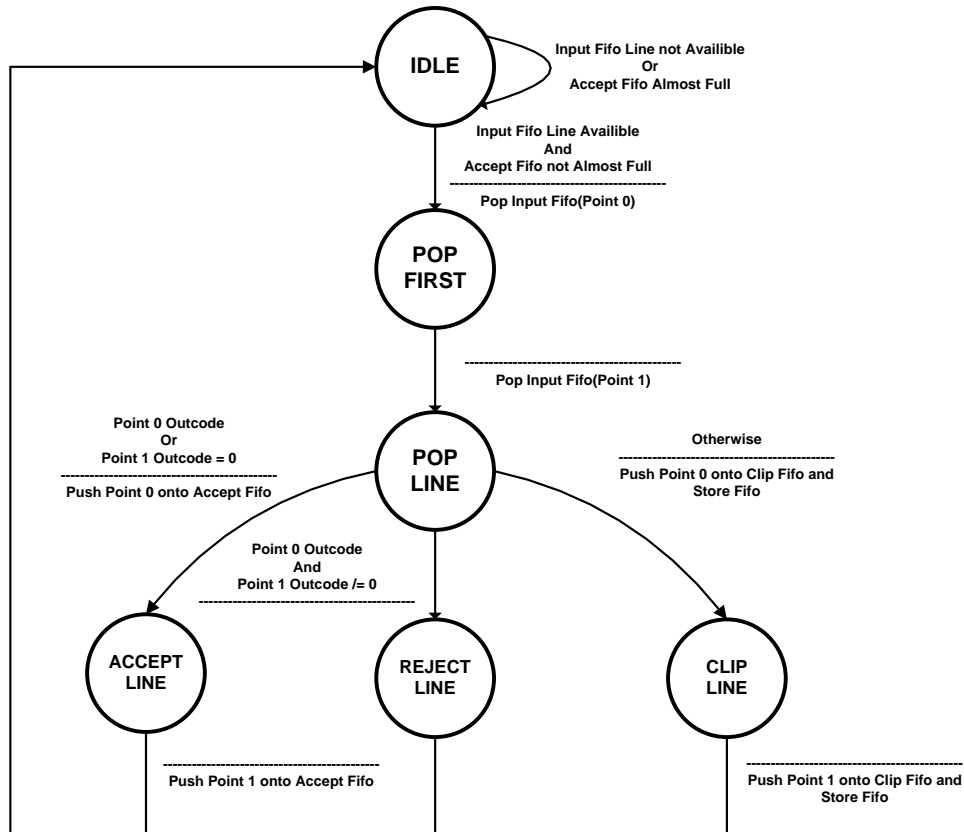


Figure 3.9: Clipping Decision Logic State Machine

In stage 3 of the pipeline, the clipping decision state machine decides which FIFOs the incoming lines should be put into. Note although this is a pipelined design, it is pipeline in the sense that it can process one line per two cycles, which is the same as one point per cycle.

In the state machine above the IDLE and POP FIRST state transitions clipping decision logic pops a line from the input FIFO once the input FIFO has stored a complete line. The outcode for each point is compared to determine which FIFO to push the line to if any. If the logical OR of the line endpoint's outcodes are zero, then the line is accepted and pushed into the accept FIFO and store FIFO. If the logical AND of the line endpoint's outcodes are not equal to zero, then the line is rejected and dropped from the pipeline. If neither of the above conditions are true then the line is pushed in the clipping FIFO where further processing will be done further down the pipeline.

In the case where a line is clipped, pipeline state four, the clip fetch logic in Figure 3.7 is responsible for popping the line's two endpoints from the clipping FIFO. Once the line is popped from the clipping FIFO, the intersection point with the viewing volume is calculated to determine the new line. While Cohen-Sutherland clipping only requires solving the intersection equations for planes/sides that the line could potentially intersect, the clip fetch block cycles through all six plane intersection calculations for 3D or four side calculations for 2D. The reason all potential intersection points are calculated is to both simplify as well as increase the performance of the clipping logic.

Section 2.2.5 derived the equations to calculate the viewing window and viewing volume intersection equations. These equations are represented below for 2D in Table 3.3 and 3D in Table 3.4.

Table 3.3: 2D Clipping Intersection Equations

| Clip Edge | Solve for t | Edge intersection equations. |
|----------------|--|---|
| $y = y_{\max}$ | $t = \frac{(y_{\max} - y_0)}{y_1 - y_0}$ | $x = x_0 + \frac{(x_1 - x_0)(y_{\max} - y_0)}{y_1 - y_0}$ |
| $y = y_{\min}$ | $t = \frac{(y_{\min} - y_0)}{y_1 - y_0}$ | $x = x_0 + \frac{(x_1 - x_0)(y_{\min} - y_0)}{y_1 - y_0}$ |
| $x = x_{\max}$ | $t = \frac{(x_{\max} - x_0)}{x_1 - x_0}$ | $y = y_0 + \frac{(y_1 - y_0)(x_{\max} - x_0)}{x_1 - x_0}$ |
| $x = x_{\min}$ | $t = \frac{(x_{\min} - x_0)}{x_1 - x_0}$ | $y = y_0 + \frac{(y_1 - y_0)(x_{\min} - x_0)}{x_1 - x_0}$ |

Table 3.4: 3D Perspective Projection Clipping Intersection Equations

| Clip Edge | Solve for t | Planar intersection equations. |
|----------------|--|---|
| $y = -z$ | $t = \frac{(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ | $x = x_0 + \frac{(x_1 - x_0)(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ |
| $y = z$ | $t = \frac{(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ | $x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ |
| $x = -z$ | $t = \frac{(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ | $y = y_0 + \frac{(y_1 - y_0)(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ |
| $x = z$ | $t = \frac{(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ | $x = x_0 + \frac{(y_1 - y_0)(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ |
| $z = -1$ | $t = \frac{(-1 - z_0)}{z_1 - z_0}$ | $x = x_0 + \frac{(x_1 - x_0)(-1 - z_0)}{z_1 - z_0}$ $y = y_0 + \frac{(y_1 - y_0)(-1 - z_0)}{z_1 - z_0}$ |
| $z = z_{\min}$ | $t = \frac{(z_{\min} - z_0)}{z_1 - z_0}$ | $x = x_0 + \frac{(x_1 - x_0)(z_{\min} - z_0)}{z_1 - z_0}$ $y = y_0 + \frac{(y_1 - y_0)(z_{\min} - z_0)}{z_1 - z_0}$ |

When a line is to be clipped, the viewing volume or viewing window intersection equations in Table 3.3 and Table 3.4 respectively are used to determine where the line intersection points. Recall that each point is assigned an outcode by the Cohen-Sutherland algorithm and that if a bit in the outcode is set to one, this indicates that the point is outside the viewing window or viewing volume. Consider the 2D example shown in Figure 3.10 on page 75. 2D points have a 4 bit outcode with each bit set to one depending on where the point lies with respect to the outside of the clipping plane. For line AD point A has an outcode of 0000 and point D has an outcode 1001. Point A having a 0000 outcode indicates that this point lies within the viewing volume. Since D has a non zero outcode, this line needs to be clipped. It can be seen from Figure 3.10 that the line needs to be clipped to line AB where point B is at the top boarder of the viewing window. To calculate B, the equations in Table 3.3 can be used to

determine the exact intersection point with the viewing window. Since point D has an outcode of 1001, point D lies both above and to the left of the viewing volume. Dependent on the testing order of the Cohen-Sutherland algorithm, the intersection check is done on the left edge or the top edge. If the top edge intersection calculation is done first the new line will be AB. The algorithm calculates point B's new outcode which is 0000. This means point B is within the viewing volume and we are done. If however, the left edge intersection calculation is done first the new line will be AC. The algorithm calculates point C's new outcode which is 1000. This is not in the viewing volume and another iteration of intersection calculation must be done. This time the calculation is done on the top edge once again yielding line AB. The algorithm calculates point B's new outcode which is 0000. Once again point B is within the viewing window.

Line EI requires up to four iterations. The first endpoint E has an outcode of 0100. Therefore the algorithm cuts the line on the bottom edge using the bottom edge intersection equation yielding the new line FI. The second endpoint I has an outcode 1010. Depending on the algorithm, either the top edge or right edge intersection can be selected to be calculated first. If the top edge is selected for clipping the new line is FH. H's outcode is determined to be 0010, so the next iteration results in clipping in the right edge. This clipped line is FG. This line lies within the viewing window and can be accepted.

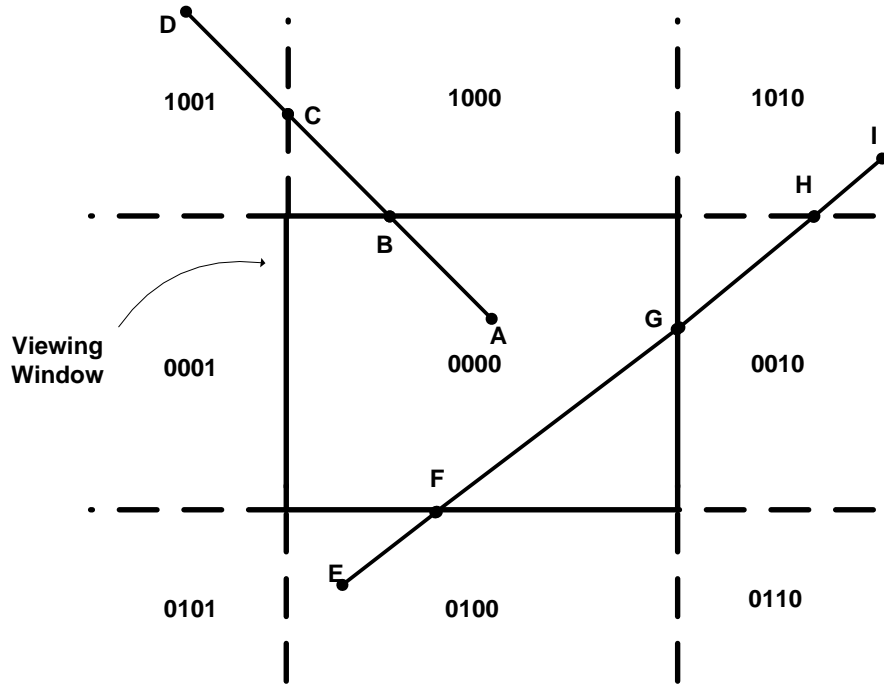


Figure 3.10: Cohen-Sutherland Line Clipping with outcodes

The above examples show that Cohen-Sutherland clipping is an iterative process that requires multiple clips before a line lies entirely within the viewing volume. In order to improve performance of the clipping algorithm all clipping viewing windows or viewing volume intersections in Table 3.3 and Table 3.4 are performed in parallel as opposed to iteratively. While this results in more logic utilization it improves the overall throughput of the clipping logic.

In addition to all intersection equations being evaluated in parallel, the intersection equations above must be completely pipelined as to not impose a bottleneck on the rest of the system. The intersection equations are composed of multiplication, addition and division floating point operations. The same pipelined floating point operations used in the matrix multiplication accelerator can be used here. Using the pipelined floating point operations, the fifth stage of the pipeline design in Figure 3.7 can be used to calculate all of the intersection equations in Table 3.3 and Table 3.4.

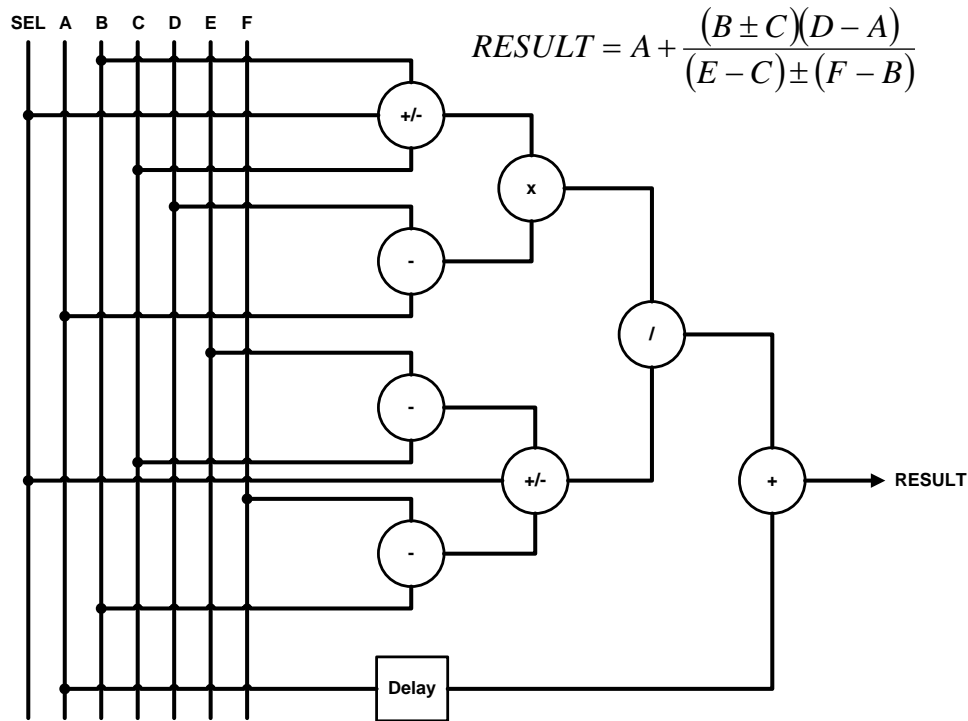


Figure 3.11: Edge Intersection calculator.

In the above figure, the select signal chooses between addition and subtraction in the +/- blocks of the clipping tree logic. For example, in Table 3.3 the bottom edge intersection equation is

$$x = x_0 + \frac{(x_1 - x_0)(y_{\min} - y_0)}{y_1 - y_0}$$

Using the edge intersection calculator, the following substitutions

$A=x_0$, $B=F=y_{\min}$, $C=y_0$, $D= x_1$ and $E=y_1$ as well as selecting subtraction for the +/- blocks can be used to calculate the bottom edge intersection of a 2D viewing window. Two edge intersection calculators are needed for the 3D engine while only one is needed for the 2D engine. These edge intersection calculators make up the Clip Tree block in Figure 3.1. The Clip Tree's internal state machine cycles through all six intersection calculations in the 3D case or all four intersection calculations in the 2D case. A byproduct of this is that it takes six cycles to complete a 3D clipping calculations and four cycles for all 2D clipping calculations. Although this fact causes clipped lines to stall the pipeline, this design decision is made for implementation simplicity

because a large amount of combinational logic is needed to determine exactly which intersection equation to use. Generating an edge intersection calculator for each intersection equation could remove the pipeline stalling, but the logical real-estate needed for this is unpractical. Since lines which are not clipped are fully pipelined and the matrix multipliers for coordinate transformation require four cycles to complete, this fact does not hurt the designs overall throughput significantly.

By using the edge intersection calculators within the clip tree new x, y, z coordinates are created that intersect with all planes of the viewing volume or viewing window. Not all of these x, y, z coordinates are needed, it depends on the original outcodes of the lines in question. First each point's original outcode is checked to see if it even needs any of the intersection calculations. If the original outcode is zero the intersection's six or four intersection calculations are unnecessary and can be discarded. In this situation the old coordinate can be used. If the original outcode is non zero than one of the intersection points calculated by the Clip Tree is the new point that will be used to complete the clipped line. To determine this another outcode generator is needed.

The new x, y, z coordinates calculated by the Clip Tree block are pushed through the next stage of the pipeline which is another outcode generator. This outcode generator is used to determine if the new intersection points lie within the clipping volume/window or not. The clip selector then looks at the old outcode by popping it from the store FIFO to determine which plane or planes the line needs to be clipped against. Once the clip plane is determined the clip selector then cycles through all the new outcodes generated. One of the new outcodes will be zero for one of the planes the line intersects. This new x, y, z coordinate corresponding to this

opcode is selected and the new clipped point is made available to the final pipeline stage, the arbiter.

One potential problem with the clipping logic is that it takes more cycles to clip a line than to trivially accept it. This fact can cause trivially accepted lines needlessly back up within the rendering pipeline. In order to avoid this situation an arbiter is added to the end of the clipping stage. This arbiter selects either a clipped line or a trivially accepted line and passes it onto the rasterizer logic. Arbitration between clipped and accepted lines is done in a fair round robin fashion. Figure 3.12 shows the functionality of the round robin arbiter.

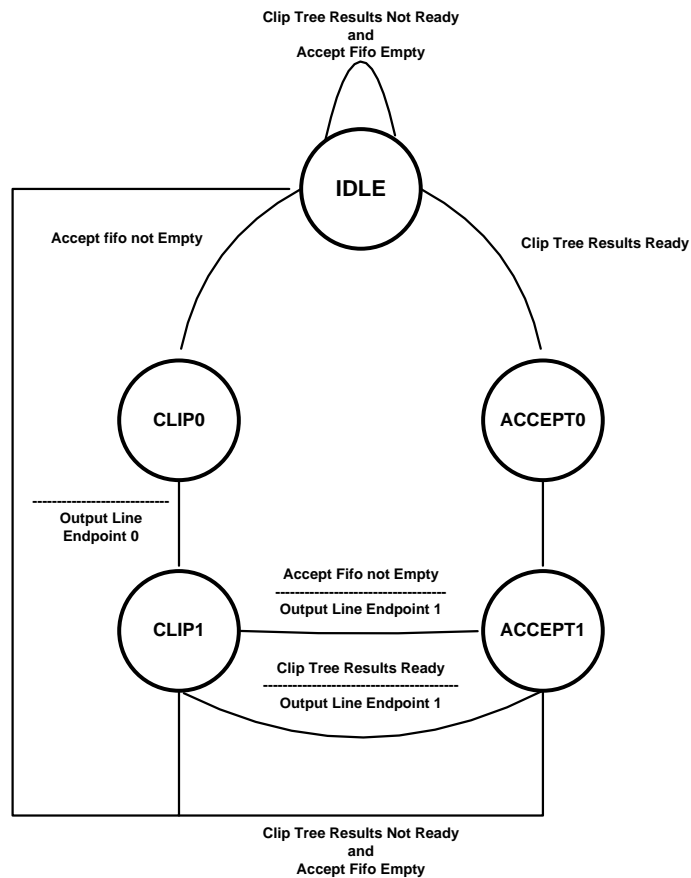


Figure 3.12: Round Robin Arbiter

Note that the arbiter waits for either the clipping tree results or for the accept FIFO to have a line ready for processing. When either is ready the arbiter outputs either the data from the accept

FIFO or the line calculated from the clipping tree logic. The state machine gives fair weight to accepted and clipped lines so that neither is starved for any substantial amount of time. The vectors output from the clipping logic are sent to the clipping coordinate transformation, where the graphics objects are projected onto a 2D plane. From there the objects are converted to screen coordinates and converted to pixels by the rasterizer.

3.1.3 Line Rasterization

Rasterization is the process of calculating each object's contribution to each pixel. Based on the specifications of the design, this design only requires wireframe rendering of graphics objects. Wireframes only consist of edges of a polygon which is nothing more than a group of lines. A good algorithm to use for line rasterization is Bresenham's line rasterization algorithm. Bresenham's line algorithm is efficient, accurate and can be easily implemented in hardware due to the fact that it is strictly based on integer math with only addition, subtraction and bit shifting. Integer operations require far less logic resources to implement making the design very compact. Also note that although the GPU pipeline uses floating point arithmetic in the coordinate conversion and clipping portions of the pipeline, the conversion to screen coordinates expresses the floating point values in a range limited by the screen coordinates. For instance if the screen is 640x480 then the floating point values will be between 0 to 640 for the x values and between 0 to 480 for the y values. The line rasterizer must round these values to the nearest pixel and the floating point values must be converted to integers. Though this introduces some error, the performance gains, area reduction and reduced design complexity justify this design decision.

Bresenham's line rasterization algorithm takes two line end-points from the 2D projection matrix transformation and fills in all the raster pixels on a display between them. The

design, shown in Figure 3.13 is a five stage pipeline which takes incoming point vectors from the output of the screen coordinate transformation and writes the appropriate pixels into the frame buffer. Once again, a pipeline design is used in order to maximize throughput enabling a high number of graphics objects to be processed every frame. Using pipelining, this particular design can process one pixel per clock cycle. Assuming this logic is clocked with a 100MHz clock, a pixel fill rate of roughly 100,000,000 pixels per second can be achieved with this architecture which matches the initial specifications of the GPU. At 60 frames per second this equates to 1.67 million pixels per frame.

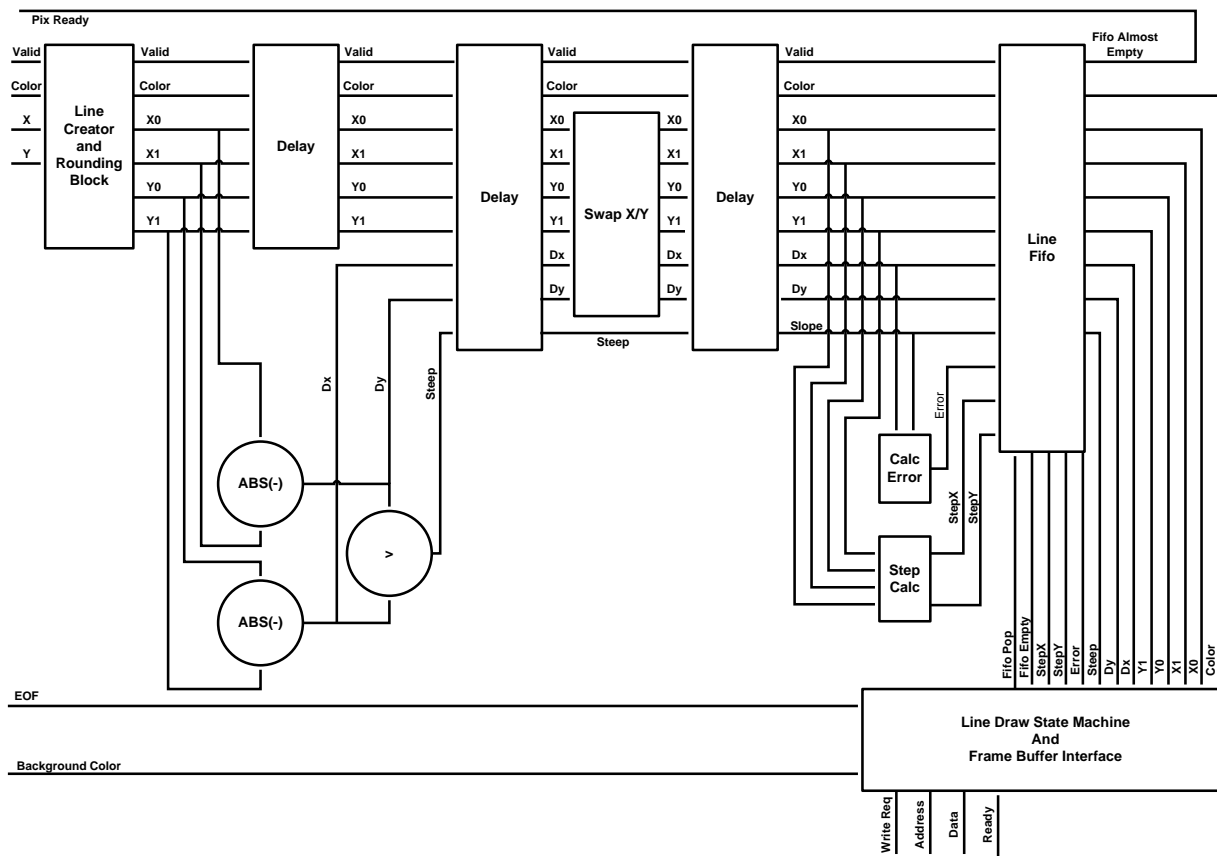


Figure 3.13: Bresenham's Line Rasterizer Design

Going through each pipeline stage of the rasterizer, the first stage, the line creator and rounding block groups two pixels from the screen transformation block to form a line. Each floating point endpoint is rounded to an integer value. Stage 2 then takes the line's endpoints which are passed to the absolute value subtractors. It is here that the change in x (Δx) and change in y (Δy) are calculated. In stage 3, the change in x and change in y are compared to determine which rate of change is larger. This value is known as the steep value and determines if the line's slope is mainly vertical or horizontal. Recall from Section 2.2.7, Bresenham's line algorithm limits the slope within a range of zero to one. In order to handle lines with slopes not between 0 and 1, the line's end points must be swapped in such a way as to force all lines within these slope constraints. In other words a line with a vertical slope must be rotated to have a horizontal slope. The most straight forward way to do this is to swap the x and y values of each endpoint in the line as is done in stage 4 of the pipeline. After the endpoints have been swapped if needed, pipeline stage 5 assigns values to the x step, y step and error flags. The error is initially set to either Δx or Δy depending on the steep value. The x step and y step flags are set and indicate if the line is going from left to right or right to left and from top to bottom or bottom to top. Lastly, the new endpoints are pushed into the line drawing state machine which handles writing the pixels for each line to the frame buffer.

The line drawing state machine, shown in Figure 3.14, takes two line endpoints from the five stage rasterization pipeline and writes the line to the frame buffer. The line drawing state machine handles two main tasks. The first is to clear the frame buffer of all the pixels stored in the last frame. The second is to then go through each line passed to the rasterizer one by one and calculate its contribution to each pixel in the frame buffer.

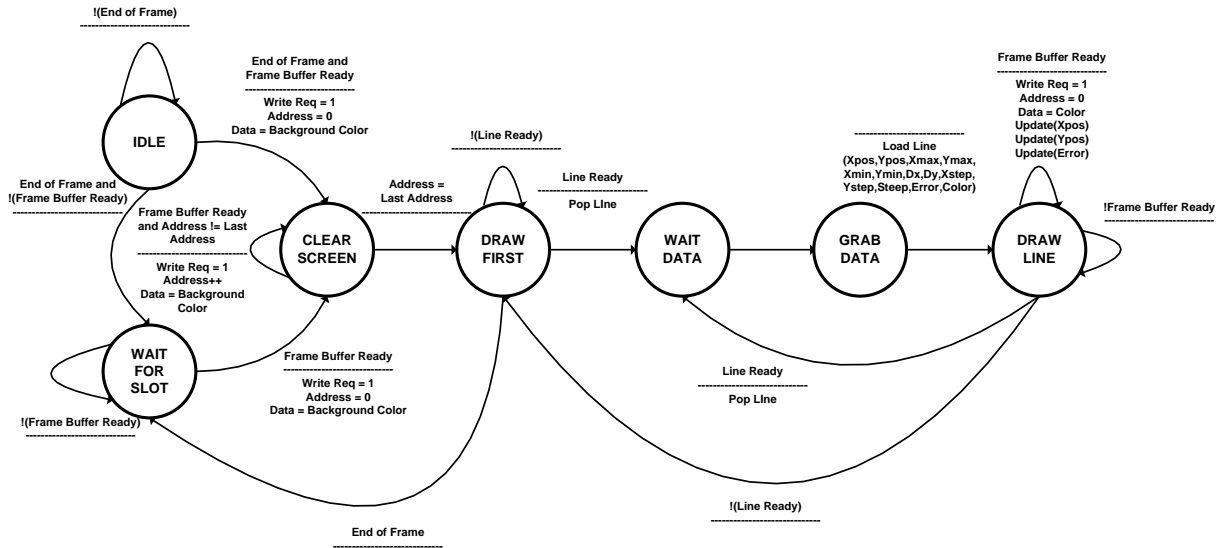


Figure 3.14: Line Drawing State Machine

Looking at the state machine above, the state machine stays idle until the display interface triggers that a frame has ended. An end of frame means that all the pixels for this particular frame have been sent to the display interface and that processing on the next frame can begin. From there the state machine waits for the frame buffer to be ready to receive data. If the frame buffer is ready, the state machine enters the CLEAR SCREEN state and begins writing to background color to every pixel in the frame buffer. Once the screen is cleared, the state machine enters the DRAW FIRST state. Here it waits for the five state pipeline in Figure 3.13 to indicate that a line is available in the line FIFO with assertion of the LINE READY signal. When LINE READY is asserted the state machine responds by asserting the POP LINE signal and then waits in the WAIT DATA state for the line data to become available. Once available the state machine enters the GRAB DATA state where the line data is parsed and registered. The registered data is then used in the DRAW LINE state. It is this state where the line's endpoints are connected with rasterized pixels. Recall from Section 2.2.7 that in the x direction that for

each pixel x is simply incremented by one. For each new x a new error is calculated. If this error is greater than the half way point between the current y position and the next y position, then y is incremented by one otherwise the y position stays the same. Remember that the notion of x position, y position and increment and decrement can be swapped this final stage since Brenham's Algorithm only works for slopes between zero and one. Other slopes may need to swap the x and y values as well as the order of the line's endpoints. Taking all this into consideration, the DRAW LINE state sends write requests with the calculated line pixels. Lines continue to be drawn until the rasterization pipeline's FIFO becomes empty indication no further lines need to be processed. The state machine then waits for another end of frame signal for the display interface where the process starts over. The next section discusses the frame buffer and display interface in details.

3.1.4 Frame Buffer and Display Interface

The last step in the graphics pipeline is driving out the rasterized frame onto the external display. The design specifications call for a display interface operating at 60 frames per second. For the display interface to operate at such speeds, it must be able to operate independently from the rest of the graphics pipeline. One way for the graphics pipeline to achieve this is to use a frame buffer. A frame buffer is a memory buffer which stores a complete frame of pixel data. Double buffering, a particular type of frame buffering, provides space for storage of two frames. The two frames are the read frame and the write frame. The read frame is read by the display interface and the pixel values read are driven to the display. The write frame is the frame being updated by the graphics pipeline. Figure 3.15 shows an example of the control for double buffering. As shown in the figure, the read and write pointers are always opposite polarity where

with each new frame these values are inverted. Generally, the read and write pointer bits are the most significant bit of the frame address in memory. That way, both the read frame and write frame get their own independent space.

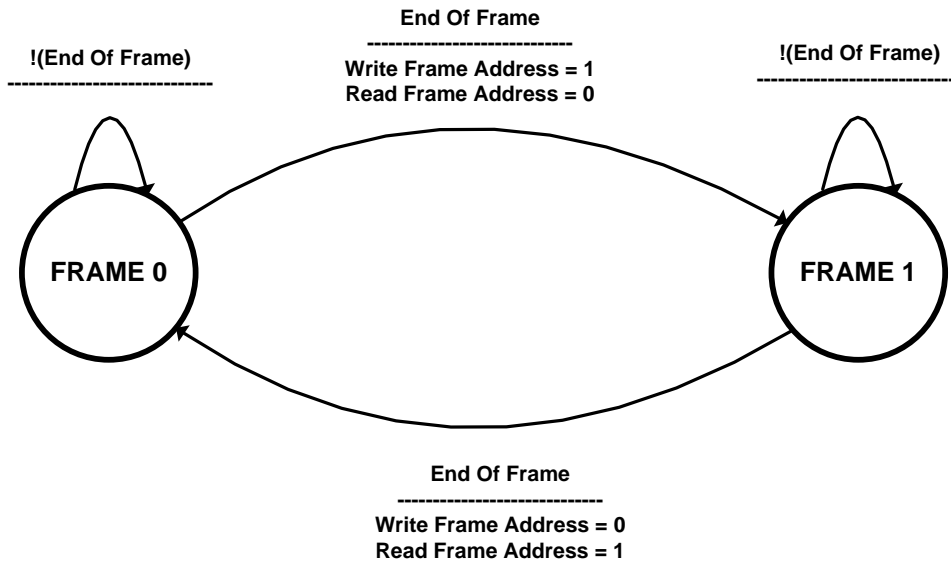


Figure 3.15: Double Buffer State Machine

By using double buffering, the graphics pipeline and display interface can access the frame buffer completely independent of each other. Without double buffering, screen artifacts can often result because the GPU is unable to complete updating the current frame before the contents of the frame are read by the display interface.

Figure 3.16 shows how the line frame buffer connects to both the line rasterizer from Figure 3.13 and the display interface.

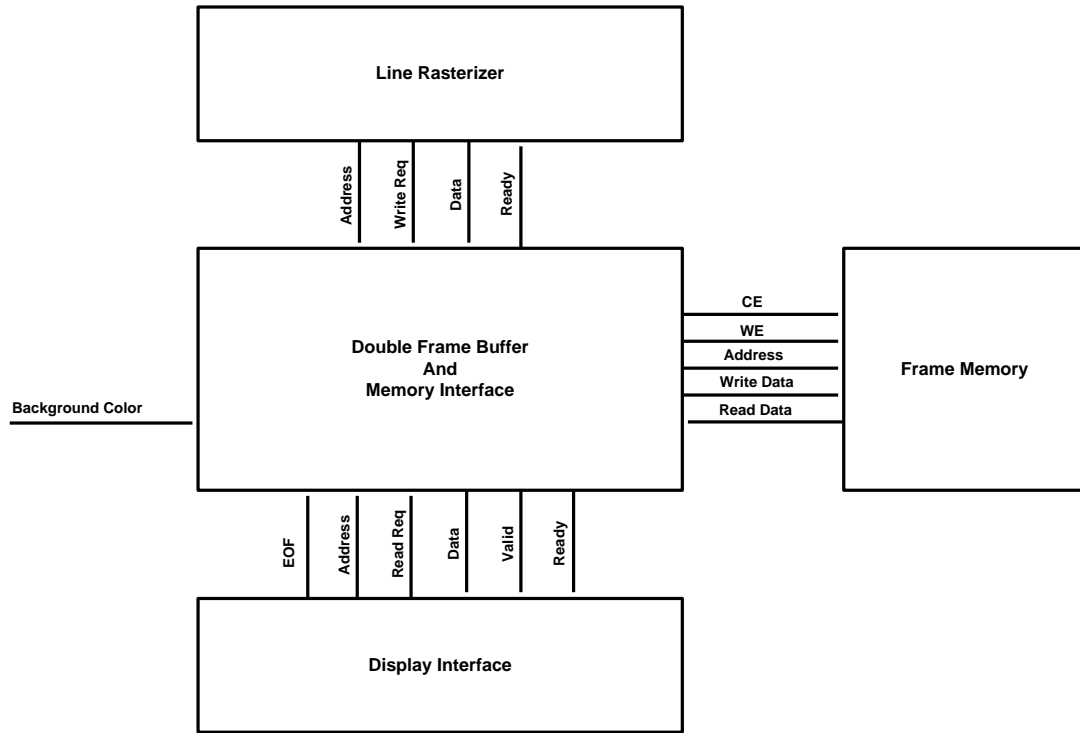


Figure 3.16: Frame Buffer Interface with Frame Memory

The line rasterizer, explained in Section 3.1.3, writes pixels to the write frame buffer as objects are processed by the GPU. On the display side, the display interface reads pixels from the read frame buffer and drives the pixels values to the display. Figure 3.17 shows the state machine which governs reading data from the read buffer.

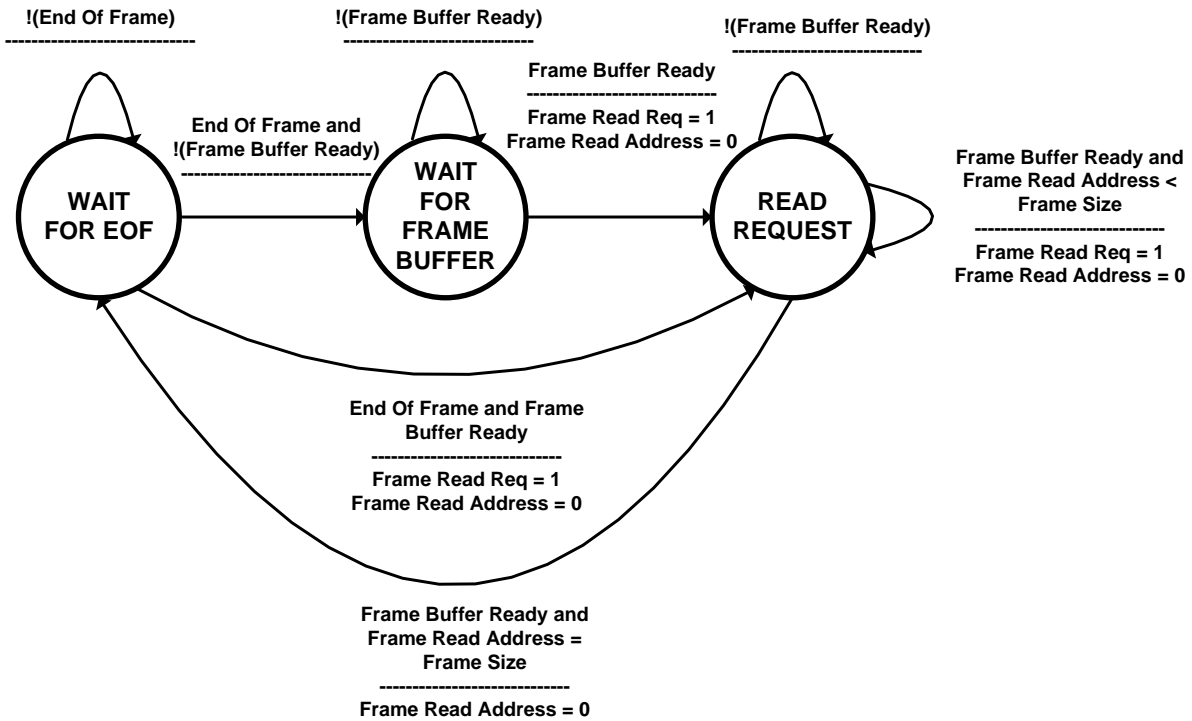


Figure 3.17: Frame Reading State Machine

As the frame reading state machine shows, the logic waits for an end of frame signal from the display interface. Upon receiving the end of frame signal, the logic waits for the frame buffer to be ready. Once ready, the display interface reads each pixel out of the read frame buffer and drives it to the display interface.

The actual display interface and type of memory are independent of this design. In Section 4.0 details are presented on both the ZBT memory and VGA interface, which are used as the frame buffer memory and display interface respectively.

3.2 CENTRAL PROCESSING UNIT

Recall the top level design shown in Figure 3.18 calls for a central control processor (CPU). The CPU handles many tasks for the GPU pipeline which are better handled in software. In addition, it provides a mechanism for initializing external components such as the display and other peripherals. The functions of the CPU will be discussed in this subsection.

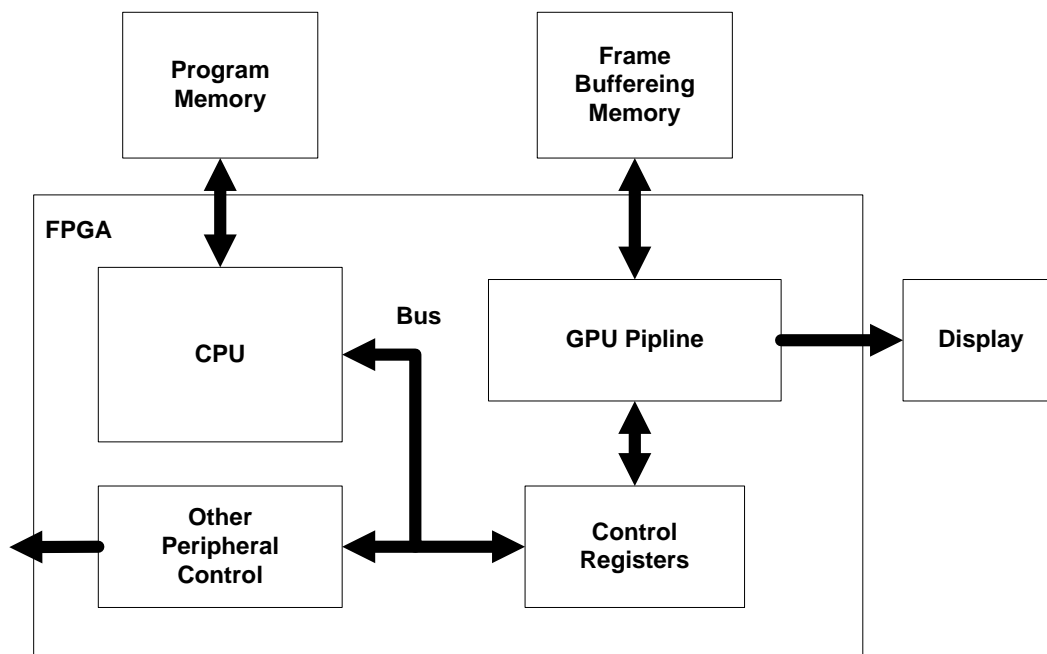


Figure 3.18: GPU Top Level Block Diagram

The GPU handles matrix multiplication acceleration for geometric transformations, clipping, rasterization and driving the rasterized image to a display. In particular, the geometric transformations required for GPU presented in Section 2.0 require many floating point mathematical operations. With geometric transformations, the elements of the transformation matrix require various trigonometric operations, multiplications, divisions and additions in order to be calculated.. While hardware could theoretically be designed to calculate the matrix elements, the trigonometric functions and calculation of matrix elements present two problems.

The first is that cosines and sine functions are not easily implemented in hardware. They either require a lookup table solution or a hardware implemented Taylor series. The lookup table solution presents a large amount of error and a Taylor series solution require many divisions and multiplications which are resource intensive in FPGAs. The second issue is that for the four geometric transformations (world, view, clipping, and screen) each has their own unique matrix elements which would require specialized hardware for each of the four transformation matrices. This adds complexity to the design and will require even further FPGA resources.

To solve these two problems above, a central processing unit (CPU) can be used calculate the matrix elements for each matrix in software. The screen and projection translation matrices only need programmed at power up due to the fact that the screen size and projection windows are constants in the design. The view matrix needs programmed once each frame as the view reference point moves through the environment. Although software solutions generally execute much slower than hardware solutions, the overhead of a software implementation does not hurt the overall performance of the design due to the limited number of times each matrix needs updated. In contrast, the world translation matrix may need updated every graphics object which does add some unwanted software overhead to design. This is acceptable though because the initial prototype is only for proof of concept. Hardware acceleration or a faster processor can be implemented later to elevate these shortcomings.

The CPU has several other responsibilities within the system. The processor handles initialization of the graphics pipeline, video interface, and other peripherals in the system. The graphics pipeline also provides an end of frame interrupt which the CPU must service. Upon every interrupt, the view transformation matrix must be updated to reflect the position of the view reference point in current frame.

3.2.1 Graphics Pipeline Control Registers

The control registers are a means for the CPU to interface to the graphics pipeline over the processor local bus. Many of these registers will be implementation dependent because external display, memory interfaces and other peripherals have different configuration registers.

An example control register circuit is shown in Figure 3.19.

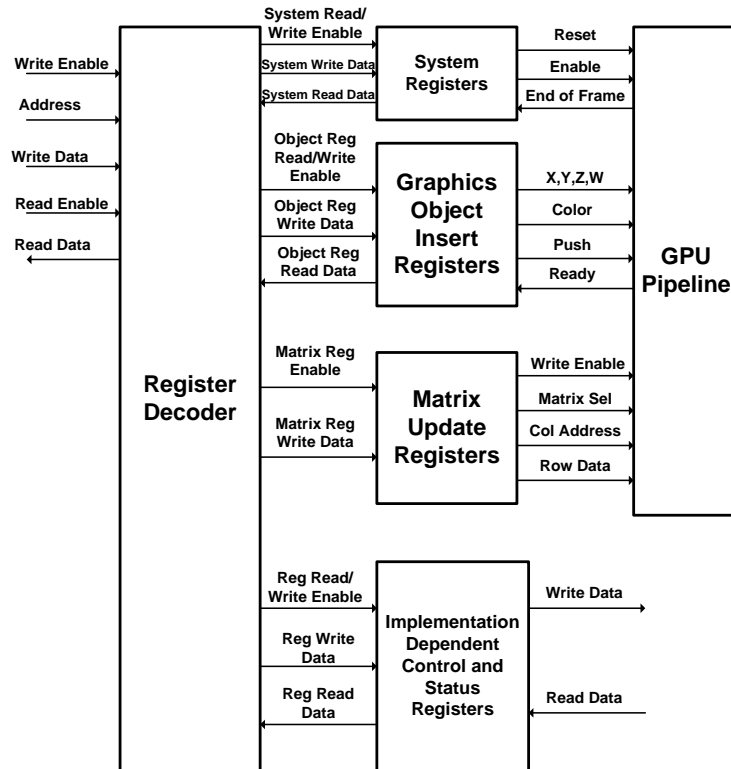


Figure 3.19: GPU Control Registers

As shown above, the GPU control interface at a minimum should have registers to pull the pipeline out of reset, initialize the pipeline, write the matrix elements, and push objects into the graphics engine. The diagram above shows system registers which can be used to reset and enable the pipeline as needed. The graphics object insert registers are used to push graphics objects on to the GPU pipeline for each frame. Another register block, the matrix update registers, enables the processor to handle transformation matrix element calculations to alleviate

the need to create custom hardware to do so. Finally, any implementation dependent registers for the system will be defined here. In Section 4.0 and Appendix A will go into detail on the these registers and any implementation dependent registers needed for both the 2D and 3D graphics pipeline.

3.2.2 Other Peripherals

Other peripherals may attach to the processor's local bus. The specifications require a video output interface such as a VGA or a DVI display interface. The CPU will handle initializing any hardware associated with the video interface. Also, any control interface such as the N64 game controller which will be used in this implementation would be initialized and controlled by the CPU as well.

This section has laid down the functional and performance specifications for the design based on the requirements for this thesis. A hybrid hardware and software design was presented which meets all the performance and functional specifications set forth in this section. The GPU pipeline uses hardware to accelerate the most computational intensive function of the graphics processor. Meanwhile, the CPU provides does additional computations for calculating transformation matrices elements and storing and sending graphics objects to the GPU pipeline. The next section will actually implement this design in a Xilinx Virtex 5 ML506 development kit with the aid of a Microblaze processor.

4.0 GRAPHIC PROCESSING UNIT IMPLEMENTATION AND TESTING

This section presents the implementation, verification and testing of the 2D and 3D graphics processing units. First, an overview of the hardware design platform selected is given in order to understand the implementation constraints. Next, the implementation of the graphics processor is discussed along with FPGA implementation results. Third, the testbench used to verify the implementation and test results are presented. Fourth, a presentation of the synthesis, mapping and place and route logic utilization results are highlighted. Lastly, details of the test software used to verify the design in real hardware is given along with actual output screens.

4.1 HARDWARE DEVELOPMENT PLATFORM

This subsection discusses the hardware development platform selected and how it meets the minimum requirements needed to implement the graphics processing system laid out in Section 3.0 . Below is a list of hardware requirements needed to implement a graphic processor system:

- Video interface to drive graphics images to a display.
- Memory storage for frame buffering.
- A control interface to manipulate the graphics environment.

- A control processor with memory for code storage to handle sending objects to the 2D or 3D engines as well as the control interface.
- Sufficient logic to implement the geometry and rasterization engines.

These requirements represent the minimum components needed to implement a graphics processing system. A hardware system must be selected that, at minimum, meets these goals.

Given the design in Section 3.0 , it can be seen that many matrix operations are required in order to implement the geometry and raster calculations needed to render a 2D or 3D scene. Commercial graphics processors generally use IEEE single precision floating point representations. Floating point arithmetic operations take a large amount of logic resources to implement in hardware. Because of this fact, it is best that a high density FPGA be used for implementation.

The Xilinx Virtex 5 family of FPGAs was the highest density FPGAs Xilinx had to offer when this project began (12). Although a smaller FPGA may be feasible, this FPGA family was selected in order to prevent any potential unforeseen hardware resource bottlenecks. In particular, the Virtex 5 SXT 50 (13) was selected. This FPGA was designed to be used particularly for digital signal processing designs. Graphics processing uses many of the same matrix multiplication arithmetic operations as digital signal processing designs making Virtex 5 SX50 ideal for graphic processing. The FPGA has 288 48-bit multiply-accumulate functions which can be used to efficiently implement the floating point multiplies and additions used in the geometry calculations of graphics processors. In addition, the Virtex 5 has 32,640 six input look up tables and flip flops which should be plenty of logic for the graphics systems needs. Also, Xilinx has embedded design tools that can efficiently implement processing elements called Microblaze. The Microblaze processor can be used as a control processor for the system.

The ML506 is a Xilinx development board that uses the Virtex 5 SXT 50 as its configurable logic chip. The board is shown below:

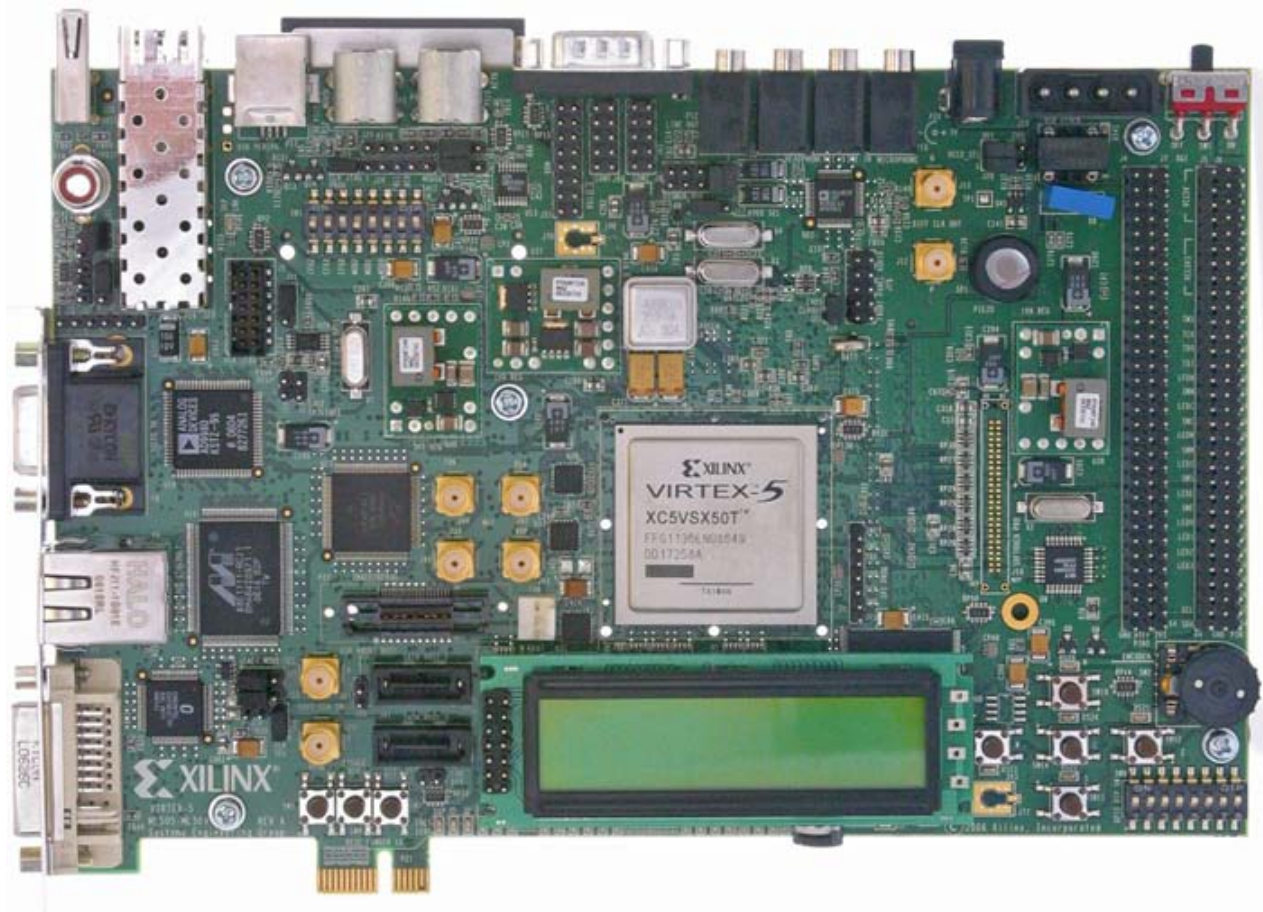


Figure 4.1: ML506 Development Board (13)

It has a DVI interface for video output. There is also a 256Kx36 ZBT SRAM for frame buffering and Z-buffering as well as a 256MB DDR2 SDRAM for code storage. In addition, the general purpose I/O can be used for any additional interfacing to the board. Lastly, it uses the Xilinx Virtex 5 SXT 50 FPGA which as discussed can provide all of the implementation logic and processing needs. Using a development board also has the advantage of having no need for a custom PCB design. Based on these reasons, the ML506 is a more than sufficient hardware platform to implement this design. Given these facts, the ML506 development board meets all the hardware requirements.

4.2 GRAPHICS PROCESSING UNIT IMPLEMENTATION

This section details the implementation of the graphics processing unit (GPU) on the Virtex 5 ML506 Development Platform. First, a discussion of the implementation of the floating point primitives needed for the graphics pipeline is presented. Second, the Microblaze CPU implementation and creation details are given. Lastly, the implementation of the graphics pipeline itself is discussed.

Recall the system design presented in Section 3.0

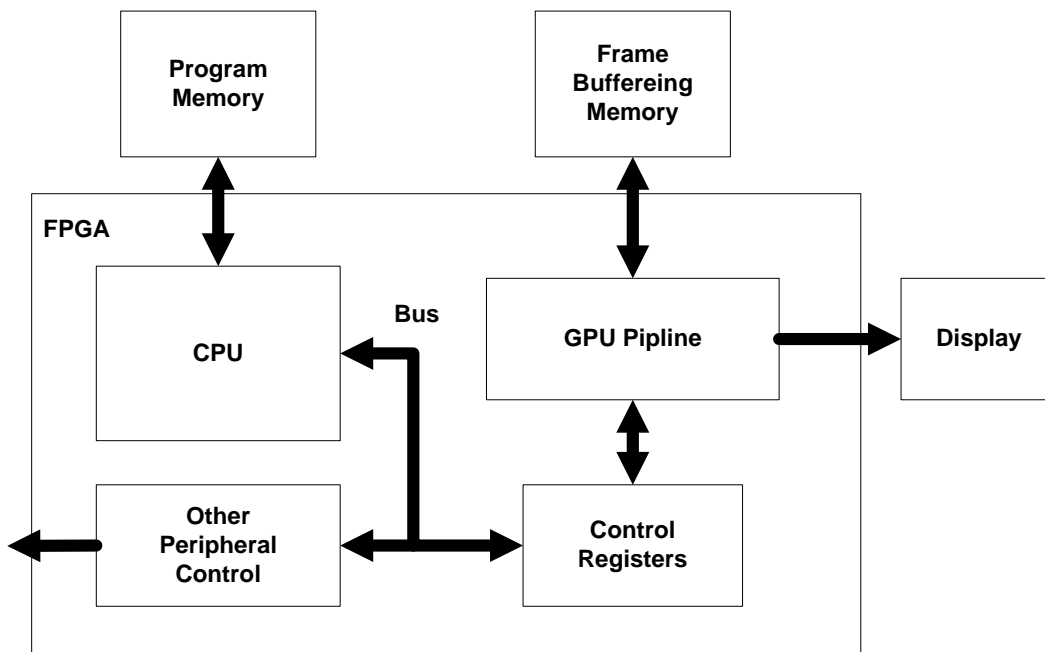


Figure 4.2: GPU Top Level Design

This design requires a GPU pipeline to be implemented along with a CPU for processing assistance. Thankfully, Xilinx's Virtex 5 ML506 Development platform provides everything needed to implement the design above. Xilinx provides a soft-core processor known as

Microblaze which can be used as the systems CPU. The FPGA fabric logic can be used to implement the floating point logic and other general purpose logic needed by the GPU. In addition the ML506 provides ZBT SRAM and DDR2 DRAM which can be used for frame buffering and program storage respectively. Finally, a DVI/VGA display interface can be used to drive a computer monitor.

Using the ML506 development kit, the system shown in Figure 4.3 was developed.

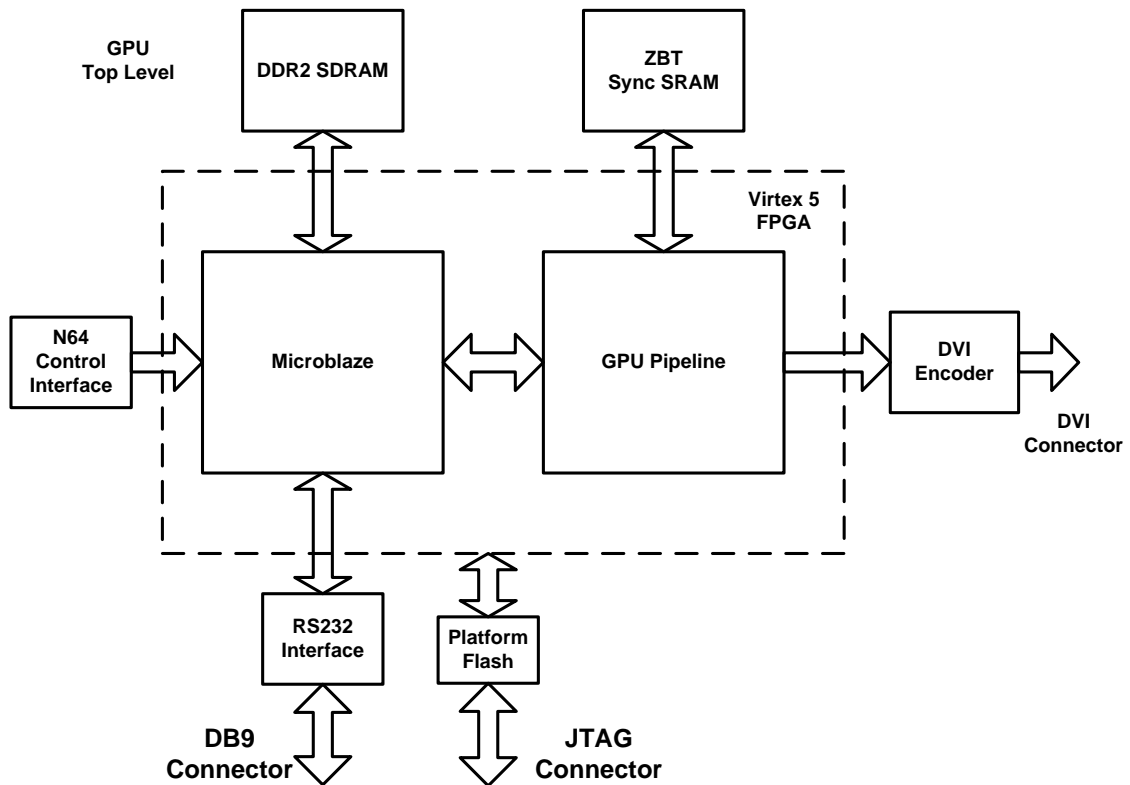


Figure 4.3: GPU Top Level Implementation Block Diagram

The Microblaze processor provides the CPU functionality discussed in Section 3.2. The Microblaze handles GPU configuration, interrupt handling, and configuring the elements within the graphic pipeline's matrix transformations. The graphics logic implements the matrix acceleration, clipping, rasterization and frame buffering presented in Section 3.1. Here floating point multiplication, division and addition cores are used to implement the computational

intensive operations needed by the GPU. The ML506 external DDR2 SDRAM is used for storing the executable code for the Microblaze processor while the ZBT SRAM is used as frame buffer storage for the GPU. A DVI encoder is used to drive a 640x480 raster display. In addition, a N64 controller is added to the system to provide a tool for manipulation of graphics objects on the computer display. The remainder of this section will go over this top level implementation in more detail.

4.2.1 Floating Point Primitives

The main mathematical primitives used in the graphics pipeline all involve floating point operations. As shown in the top level diagram, the ZBT SRAM will be used as the frame buffering memory. Due to the limitations of the ZBT 36 bit memory width, the floating point cores are not standard IEEE754 32bit single precision but custom 18bit floating point cores. This is because the ZBT memory is only 36 bits wide. The pixels are stored as 18 bit color values in order to leave room in the frame buffer to store both an 18 bit floating point z values, This will enable future implementation of Z-buffering.. Because of this decision, all floating point cores within the design use this custom 18 bit floating point format. An additional ZBT memory device could alleviate this constraint and allow the use of full 32 bit floating point numbers, but, since this is a prototype, this level of precision is acceptable for proof of concept.

The figures below shows examples of both and IEEE 754-1985 standard single precision floating point number (14) as well as the custom 18 bit floating point numbers used in this design.

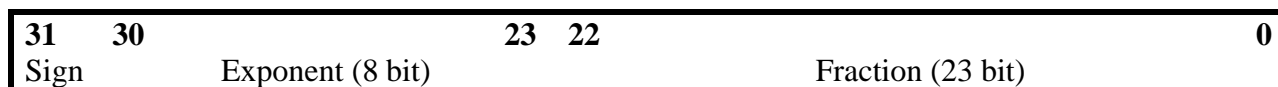


Figure 4.4 : IEEE 754-1985 32 Bit Floating Point Number



Figure 4.5 : Custom 18 Bit Floating Point Number

A floating point number is calculated by the formula below.

$$V = -1^S \times 2^E \times 1.F$$

Equation 4.1: Floating point calculation.

Where V is the floating point number, S is the sign, E is the exponent and F is the fraction.

There are several different types of floating point operations used in this design. In particular the operations needed are addition, subtraction, multiplication, division, compare, 32bit to 18bit floating point conversion, and 18bit floating point to fixed point conversion. The 32 to 18 bit conversion is needed because the Microblaze uses IEEE754 32bit single precision format. When the floating point values are passed to the GPU, they need to be converted to the 18 floating point format for processing by the 18 bit floating point cores. Additionally, in the rasterization stage, the 18 bit floating point numbers must be converted to integers requiring the 18 bit floating point to fixed cores.

Floating point calculations, either using IEEE754 single precision 32 bit or custom 18 bit, take quite a bit of logic resources to implement. These functions are also very computationally expensive in terms of software CPU cycles if a software implementation is employed. Either way, floating point operations are usually required in graphics processing due to the large

quantization errors that can be caused by fixed point or integer representations. This is especially true with higher resolution displays.

Thankfully, Xilinx provides fully pipelined floating point cores that consume a reasonable level of hardware resources on the Virtex 5 FPGA (around 100-300 look up tables depending on the operation). These cores can be customized with a variable number of pipeline stages that provides an area verses performance tradeoff.

The floating point cores are generated using the Xilinx Coregen IP Generator (15). The floating point Coregen window is shown in the figure below.

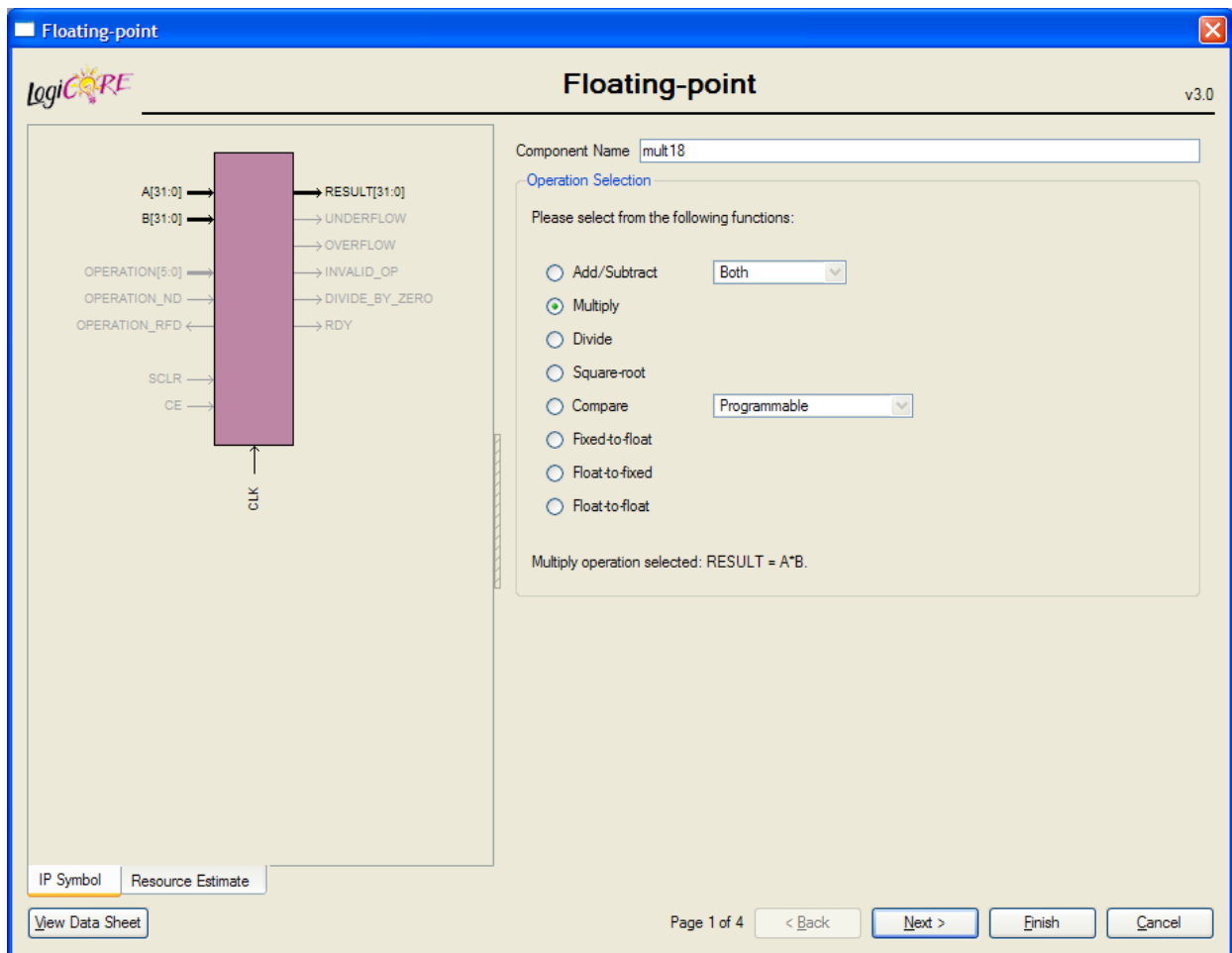


Figure 4.6: Xilinx Coregen Floating Point Operation Selection Window

From this window the function to be created can be selected. In the above instance, multiplication is selected. Once the operation is selected, a precision type needs to be selected. In the below example, the creation of a custom 18 bit floating point precision type is selected. The table below shows this selection.

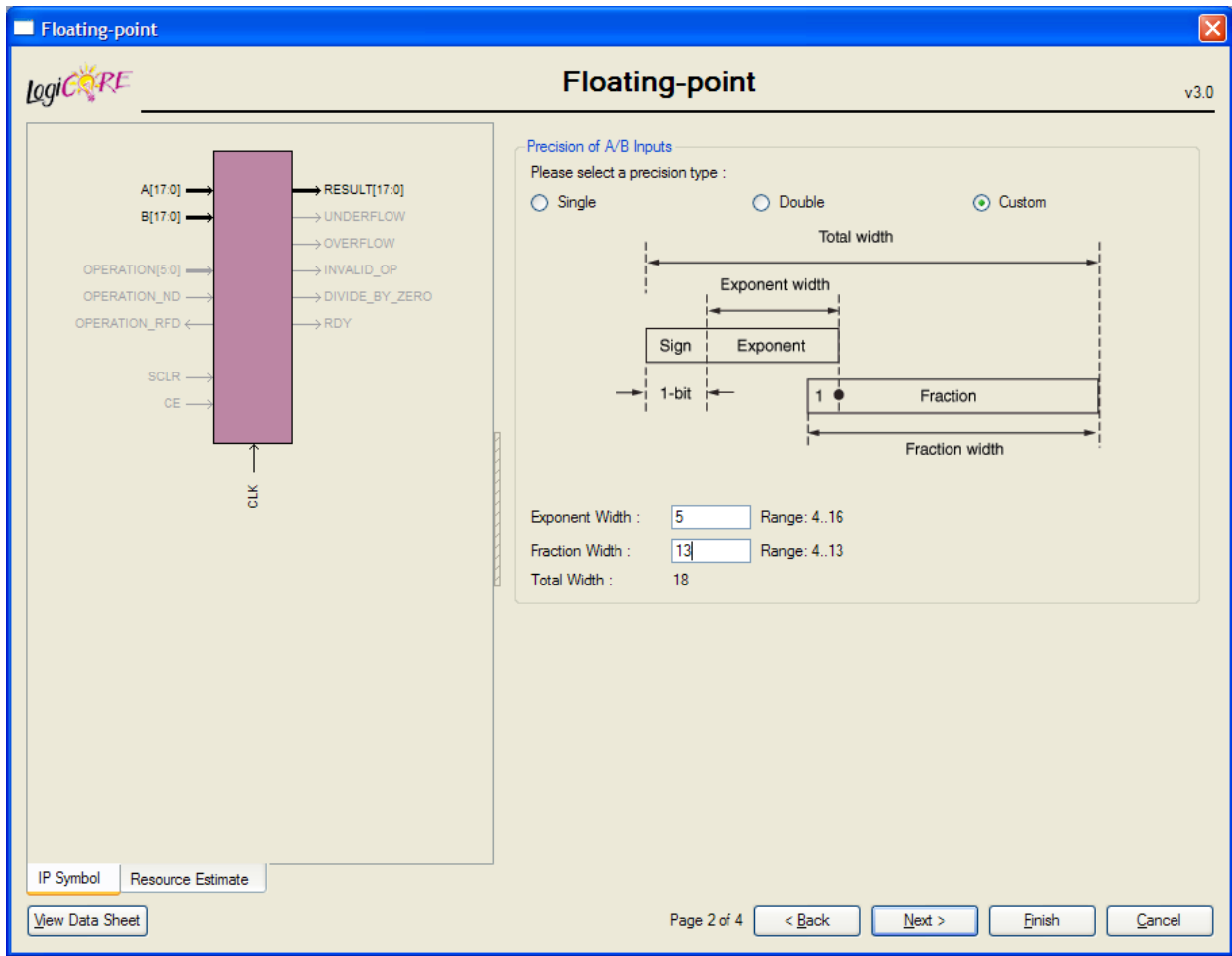


Figure 4.7: Floating Point Precision Selection Window

This precision type has one sign bit, a 5 bit exponent, and a 12 bit fraction. Note that the above diagram shows a 13 bit fraction, which is because the Coregen tool includes the sign bit in its fraction width calculation. The true fraction length is 12 bits. Once again, this 18 bit custom precision is selected due to the ZBT memory size limitation. If more memory was present IEEE754 single or even IEEE754 double precision could be used.

Lastly, Coregen allows the user to adjust the cores to either maximize area or performance. DSP48 elements can be used to reduce LUT count for addition, subtraction and multiplication (16). The DSP48 elements also increase the overall performance of the cores by reducing the number of latency cycles needed for addition and multiplication to complete. DSP48 blocks should be used when available. DSP48 elements are built in multiply and accumulate functions which can be used to perform complex math functions without the use of the logic within the FPGA fabric. Using these blocks improves area efficiency by reducing the number of logic resources used by the floating point cores as well as improves overall computational performance allowing for fewer cycles of latency in the floating point calculations. Note that DSP48 elements are limited resources with only 288 DSP48 elements on the Virtex 5.

4.2.2 Microblaze Implementation

The Microblaze Central Processor Unit handles several tasks within the GPU system. Its primary function is controlling the GPU pipeline logic by handling initialization, serving end of frame interrupts, and sending objects to the GPU pipeline input FIFO. It is also responsible for pooling the control interface logic (N64 controller) and decoding the incoming controller commands. The CPU is implemented using Xilinx's Embedded Development Kit and included soft core processor Microblaze.

The Microblaze processor is a 32-bit soft core processor optimized for Xilinx FPGA implementations. The processor is a reduced instruction set computer (RISC) and can be used to implement a wide array of software applications. A block diagram is shown below:

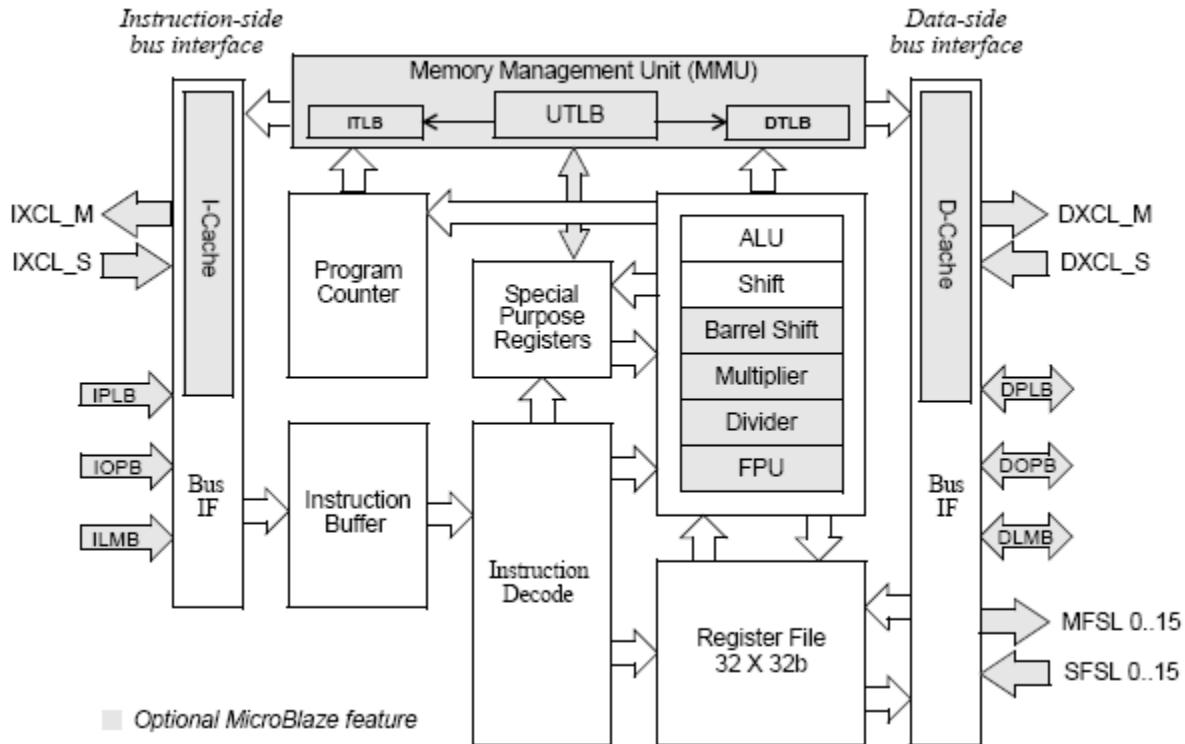


Figure 4.8: Microblaze Core Block Diagram(17)

As can be seen above, Microblaze uses a Harvard memory architecture where data and instruction accesses are stored in separate memory addressing space. Microblaze does not separate accesses to I/O peripherals and memory, it instead uses memory mapped I/O. Microblaze has three separate interfaces for memory accesses. The first is the processor local bus (PLB) which is most commonly used for access to slower external peripherals. Second is the local memory bus (LMB) which is used for reading and writing local block ram. This can be used for cache memory with a single or two cycle access latency depending on the area/performance constraints. Lastly, the Xilinx CacheLink which interfaces to the data and instruction cache controllers if caching is enabled.

Thirty-two 32-bit general purpose registers are present for software use. In addition, 32-bit instructions with two addressing modes. The processor can be configured to provide either a three stage or five stage pipeline depending on if minimum area or maximum throughput is

needed respectively. A five stage pipeline is used in this design. Microblaze also supports one external interrupt input which is used to break normal execution for high priority interrupt service routines. Virtual memory management, caching, floating point and various other instructions are optional and can be enabled or disabled depending on available resources and functionality needs. In this design, both caching and floating point logic is enabled. Caching improves code execution time while the floating point logic is useful for calculating the matrix elements in the transformation matrices.

The Microblaze processor is used to run system software for the graphics processor and interfaces to external memory as well as external peripheral. To store this software code, the Microblaze processor interfaces with external memory via the Multi-Port Memory Controller (MPMC) provided by Xilinx. The MPMC is a fully programmable memory controller provided by Xilinx that supports double data rate (DDR) or single data rate (SDR) memories. Up to eight ports can be enabled for use with data widths up to 64 bits. The eight ports can be configured for fixed, round robin or a hybrid of fixed and round robin scheduling. It supports various physical interfaces such as DDR2 SDRAM which is used in this implementation. The MPMC is a convenient choice because it handles all the refreshes, timing and clock generation for the DDR2 SDRAM ram. Figure 4.9 below presents the MPMC layout in the GPU design showing the Microblaze interfacing to the MPMC's XCL and PLB ports as it is used in this design.

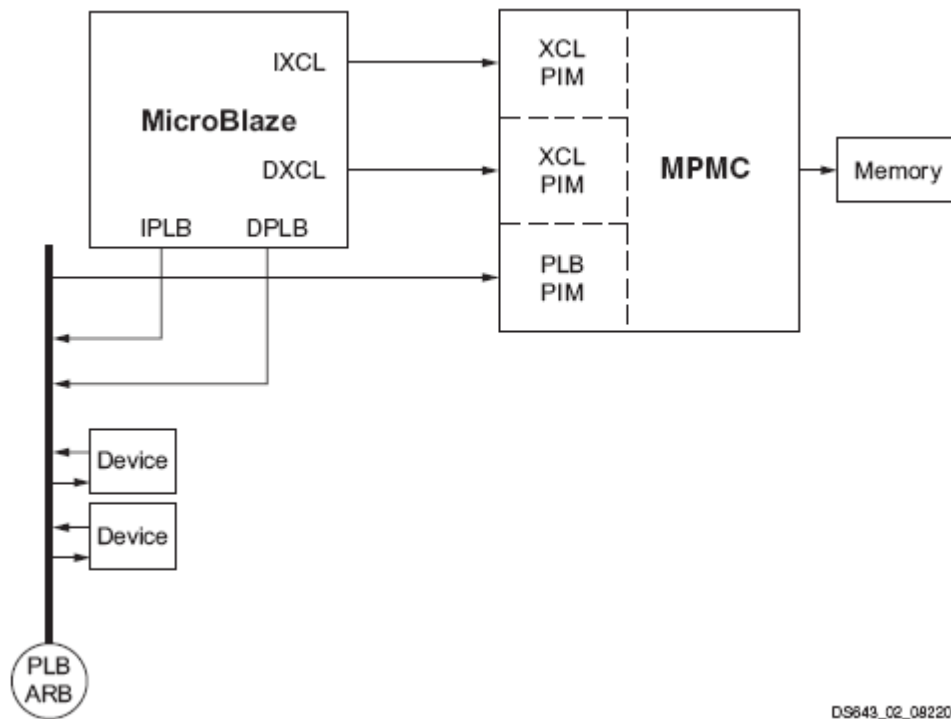


Figure 4.9: Multi-port Memory Interface Layout (18).

The GPU pipeline and other peripherals are connected to Microblaze processor via the Processor Local Bus (PLB). The PLB provides fairly arbitrated access to all PLB masters and slaves. This bus is shown in Figure 4.10.

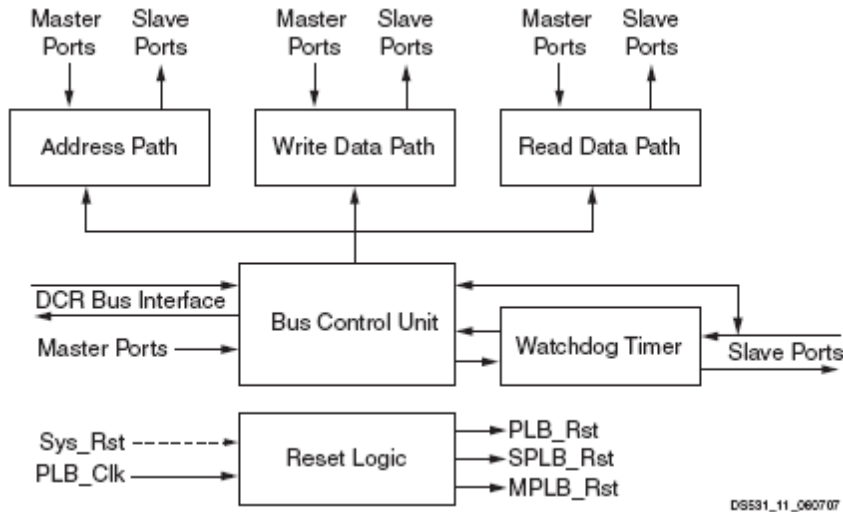


Figure 4.10: PLB Block Diagram(19)

The PLB's bus control unit provides arbitration for up to eight master devices on the bus. This logic also provides the control for steering data to the proper slave or master. In addition, the bus control unit also interfaces with a watchdog timer. This timer is used to determine if a request to a slave device has taken too long. In this case an exception is raised and needs to be handled by software. There are also separate read, write and address paths each with the muxing logic used to steer address or data to the proper master or slave device.

The PLB slaves consist of the graphics pipeline, N64 Controller, as well as various other IP cores such as the DVI IIC controller, interrupt controller, RS232 serial controller, flash memory and the DDR2 Multi-port Memory Controller. The diagram below shows the top level block diagram of the entire Microblaze system.

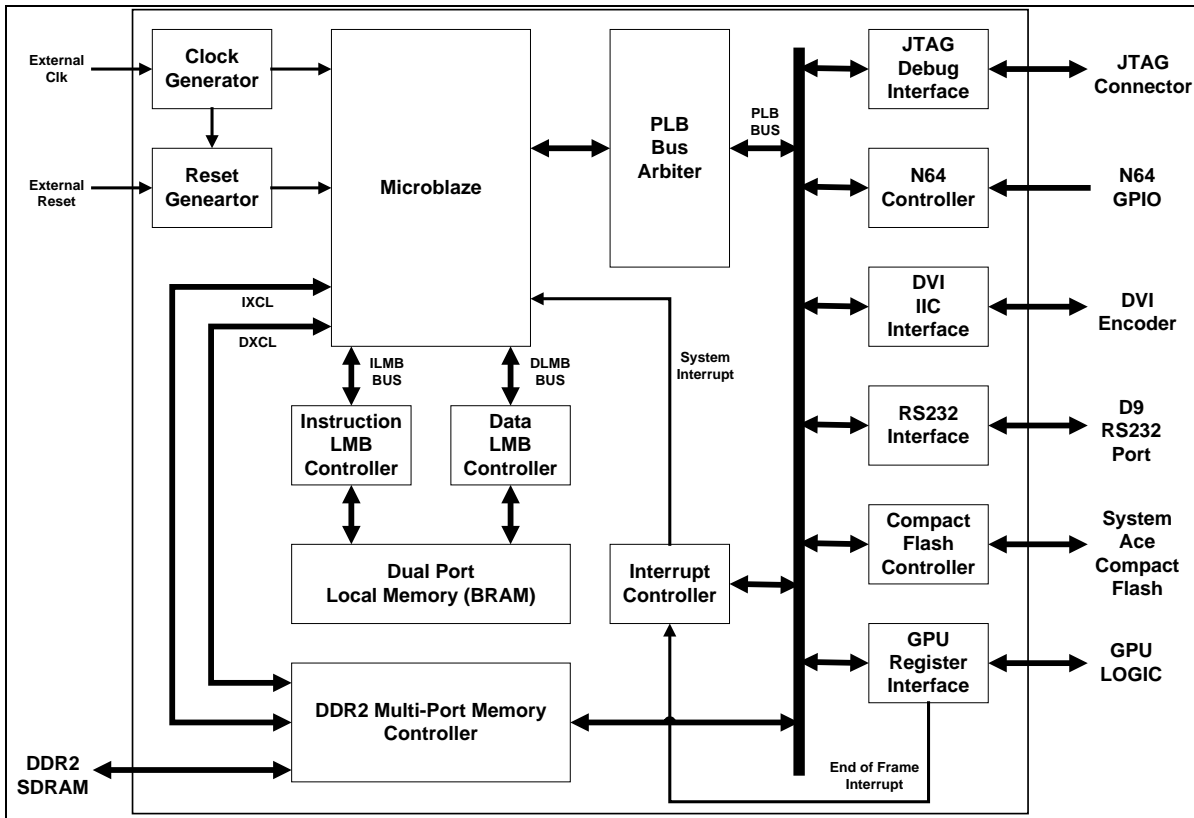


Figure 4.11: Microblaze System

Xilinx Embedded Development Kit (EDK) is a tool that helps in the generation of embedded processors on FPGA devices. The system above was mostly generated by EDK with the addition of some custom VHDL. EDK provides a tool called Base System Builder that aids in the creation of embedded systems. Because the Virtex 5 ML506 board is a board provided by Xilinx, EDK already has a pre-canned system with all the pinout files, memory controllers, bus arbiter, serial debug interfaces and interrupt controller. The tool also generates templates for PLB master and slaves to be added to the PLB bus. In this design, the GPU register interface, N64 controller interface and DVI IIC interface all need to be made using a custom design.

4.2.2.1 Base System Builder

Xilinx's Embedded Development Kit provides a Base System Builder (BSB) that provides the necessary configuration files for many popular Xilinx FPGA based development boards. Thankfully, the BSB has a configuration for the ML506 development board. Using the BSB, all the timing, area and pinout constraints for the design can be automatically generated.

To start Base System Builder one must simply start Xilinx's EDK as shown in

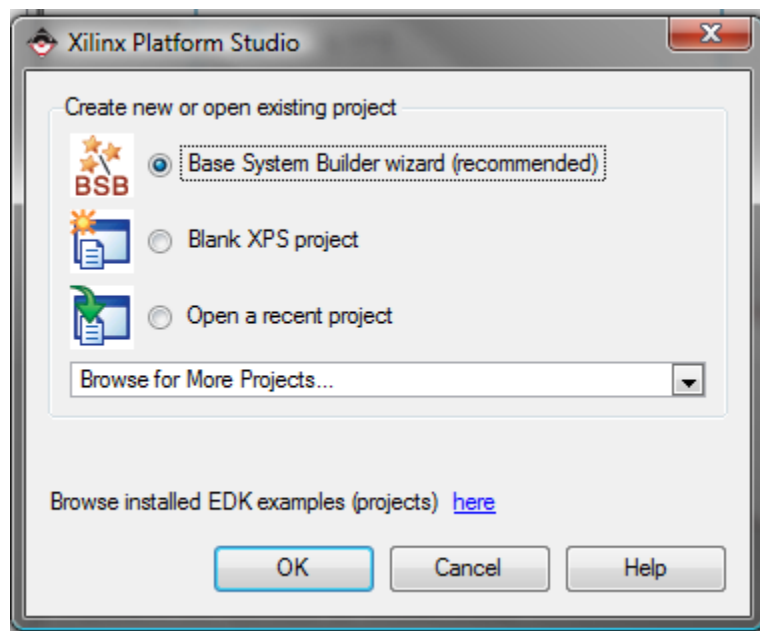


Figure 4.12: Xilinx Platform Studio's Project Opener

Selecting Base System Builder creates a directory for all of EDK's hardware, software, and configuration files. Once a directory is selected the user is welcomed by the Base System Builder tool shown in Figure 4.13. This is a new design so the option to "I would like to create a new design" is selected.

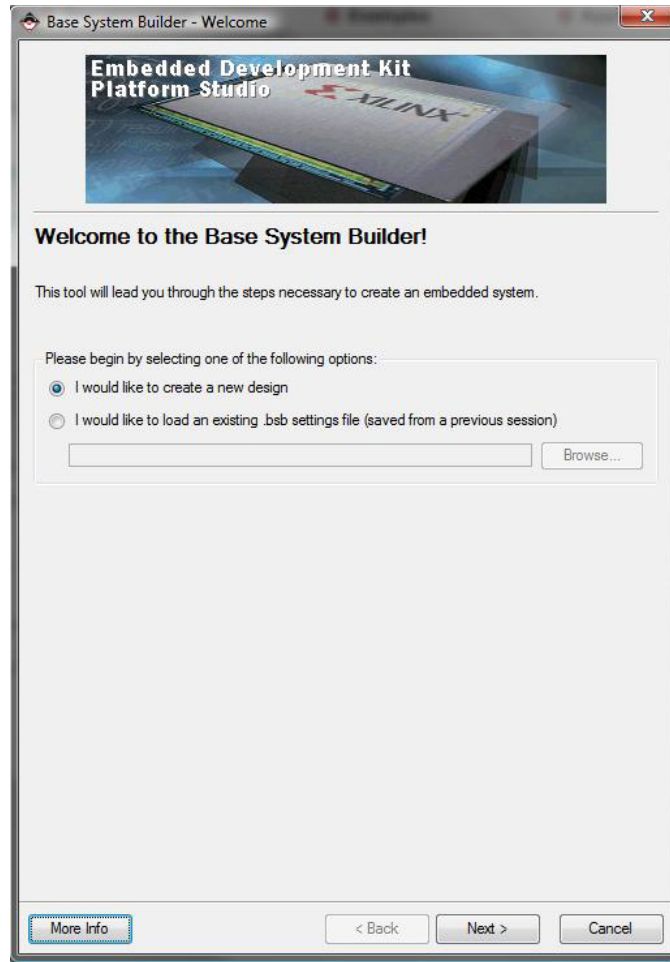


Figure 4.13: Base System Builder Welcome Window

Since the Virtex 5 ML506 Development board is a common platform, as said before, Xilinx already has all the board files needed to implement the system. To implement the system all that needs to be done is select the proper dropdown bars in Figure 4.14.

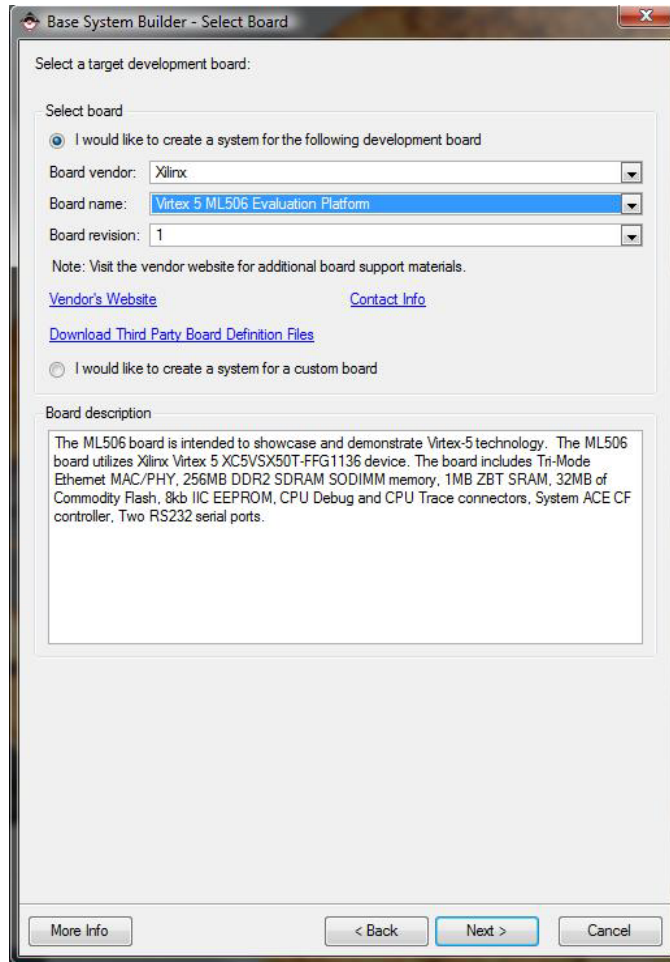


Figure 4.14: Base System Builder Board Selector

Once the proper vendor, board and revision are selected the Base System Builder requests which processor type is to be used. Many Xilinx FPGA has build in PowerPCs, the ML506 does not so the only choice is to use the soft-core Microblaze which is implemented in the FPGA fabric. The processor selection process is shown in Figure 4.15.

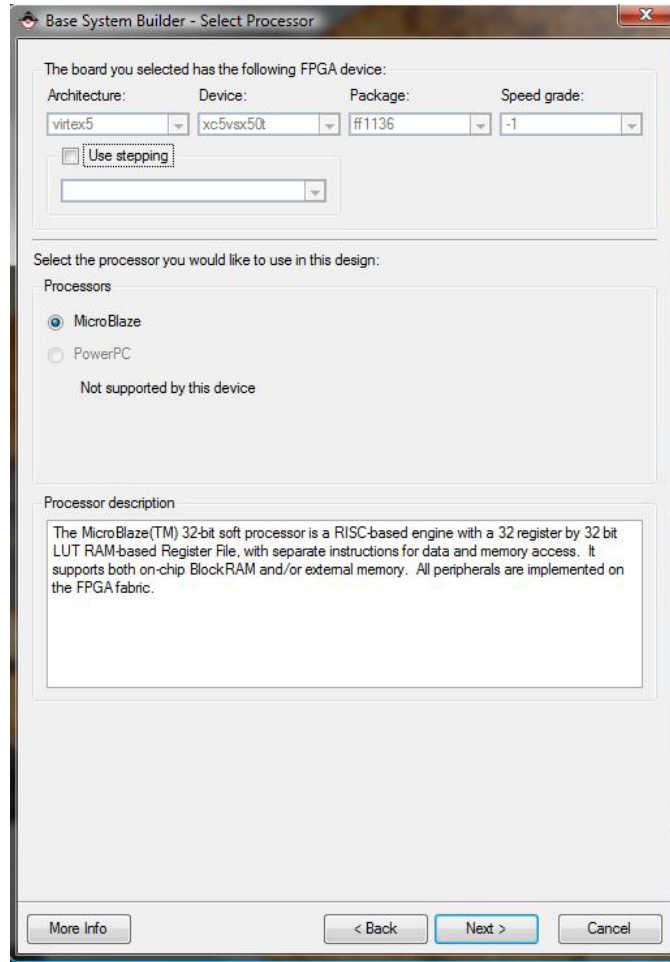


Figure 4.15: Base System Builder Processor Selector

The next step in the Base System Builder is to configure the Microblaze processor, here the processors reference clock frequency, bus clock frequency and local memory size are selected. In addition, either caching or the floating point unit can be enabled. The processor configuration is in Figure 4.16.

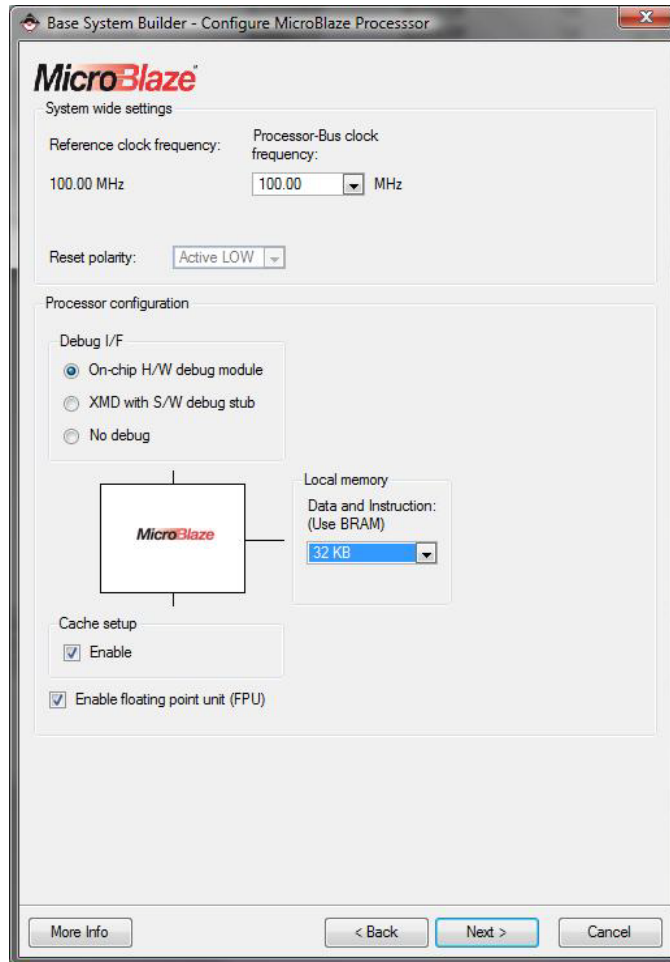


Figure 4.16: Base System Builder Microblaze Processor Configuration Window

Note that both a processor and bus speed of 100Mhz is selected. The bus speed was picked to match the processor speed to eliminate the need for any clock domain crossing logic reducing the LUT count. Also notice that a small 32KB BRAM is created for local program storage if needed. This memory is also used to load a standard bootloader created by Xilinx for debugging purposes and can be used to support more complex bootloaders in the future. In addition, the caching logic and floating point units are enabled. Caching improves overall processor performance, while the floating point units are especially useful for calculating the transformation matrix floating point values.

After configuring the Microblaze processor’s speed, cache and floating point setup, the next step is to configure the various IO interfaces that connect to the PLB bus listed in . This takes place in four steps. In the first step a single uart with a baudrate of 115200 bits per second is created. This PLB slave is used by the system as a standard input and standard output device. The standard input and standard output device is used to send debug messages to a serial console that is useful in detecting GPU status conditions and debugging the GPU. In the second step no IO devices are created because these devices are not needed by the system. In particular the ZBT SRAM memory is used by the GPU pipeline negating the need for an SRAM memory controller. Step three instantiates the Multi-Port Memory Controller (MPMC) that interfaces to the DDR2 SDRAM. The DDR2 SDRAM is used for storage of software code for use by the Microblaze. Lastly, step 4 the SysACE Compact Flash controller is created. While not need by the GPU, in the future graphics objects and software code could be stored on this flash.

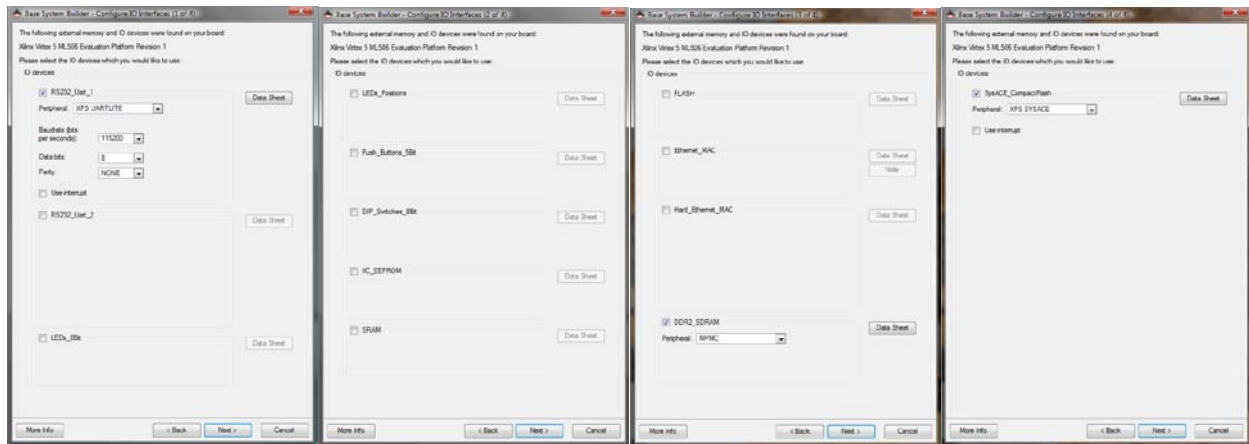


Figure 4.17: Base System Builder IO Interfaces Configuration Windows

The Base System Builder also handles creating the cache controller which interfaces to the Microblaze processor and the MPMC. The window in Figure 4.18 provides a means to configure the instruction and data cache sizes and enable them for the DDR2 SDRAM.

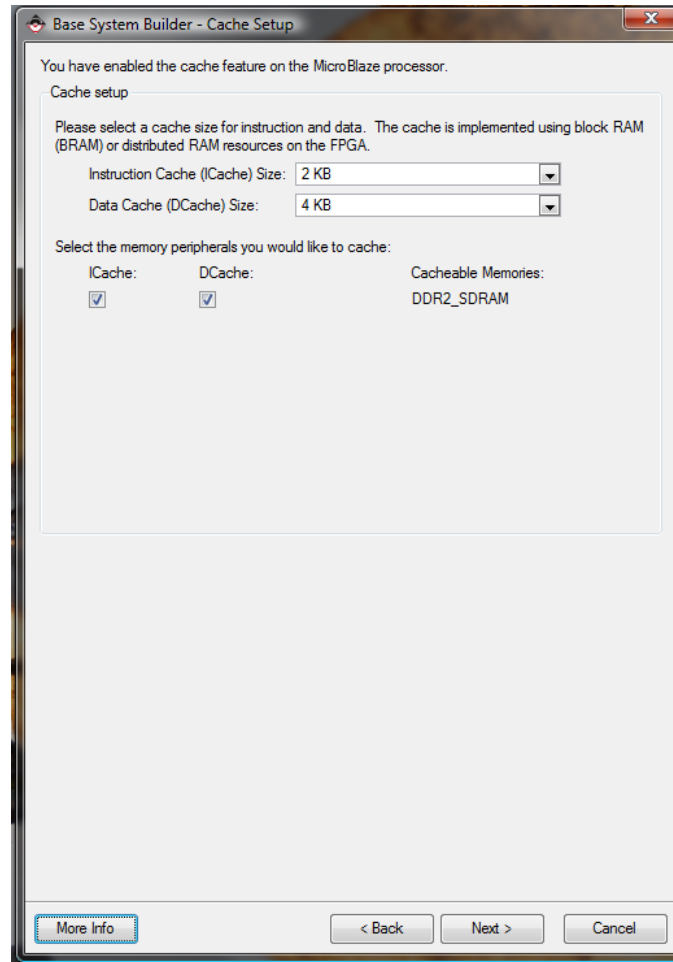


Figure 4.18: Base System Builder Cache Setup Window

Lastly in Figure 4.19, the Base System Builder selects the standard input and standard output devices. The RS232 Uart created in the IO creation section is selected to be used for debug messages. In addition, software code which can be used to test the DDR2 SDRAM is generated.

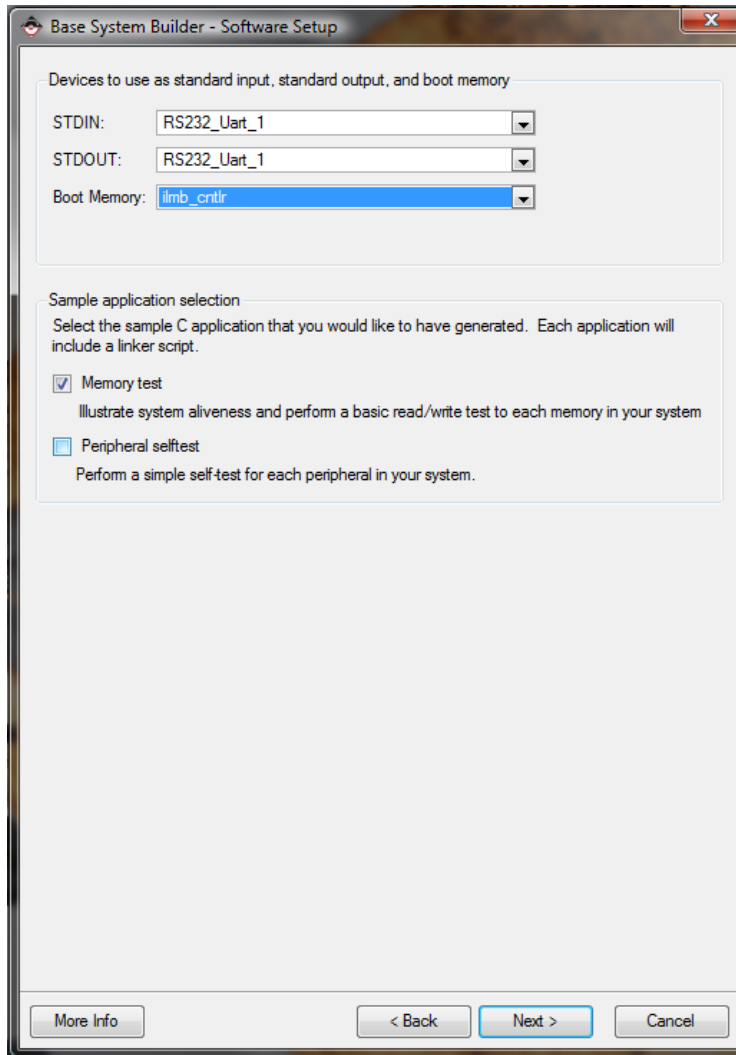


Figure 4.19: Base System Builder Software Setup Window

Now that the system is created, the Base System Builder provides a summary for the system which is created shown in Figure 4.20.

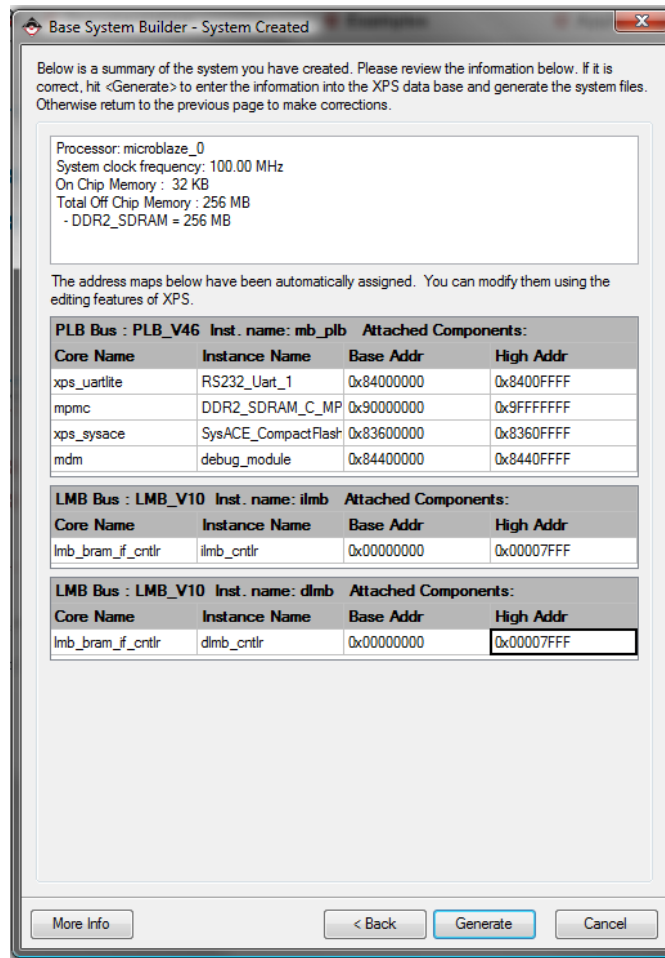


Figure 4.20: Base System Builder System Created Window

To create the base system, visualized in Figure 4.21, the user clicks on the generate button. By doing this, the Microblaze processor, local memory BRAM, PLB Bus, interrupt controller, clock manager, RS232 debug serial interface, Compact Flash memory controller, and DDR2 DRAM MPMC memory controller is created. This system includes the pin out files, drivers for the peripherals and memory test software.

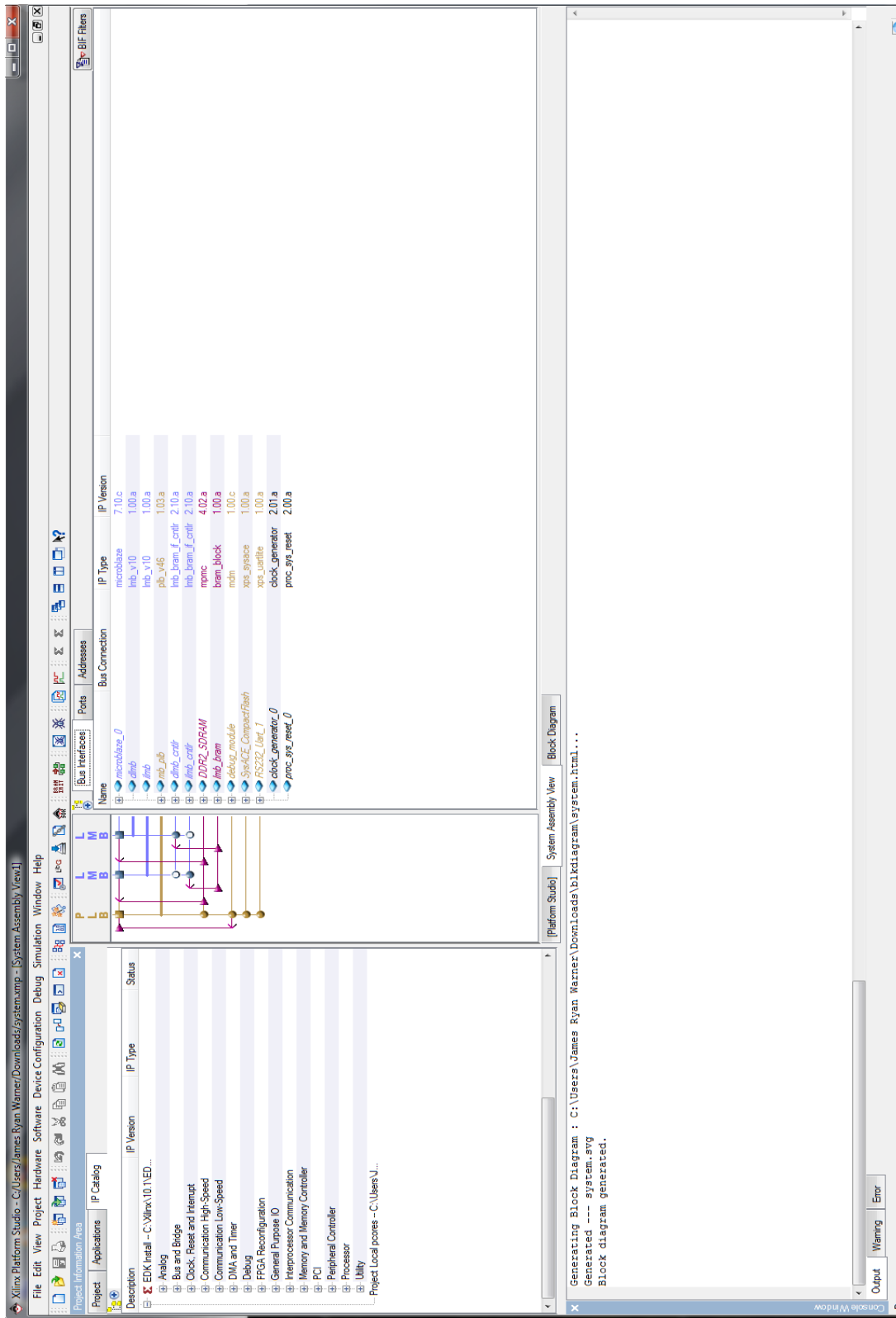


Figure 4.21: Original Base System without Custom Peripherals.

The next step is to add the custom peripherals that will connect to the Microblaze systems PLB bus. Xilinx’s EDK provides a tool to help create a VHDL template for interface to PLB. The GPU requires three slave peripherals. Figure 4.22 shows how to create a PLB slave. The key checkbox is the “User logic software register” which creates a template for a register controller within the PLB slave which all three peripherals need.

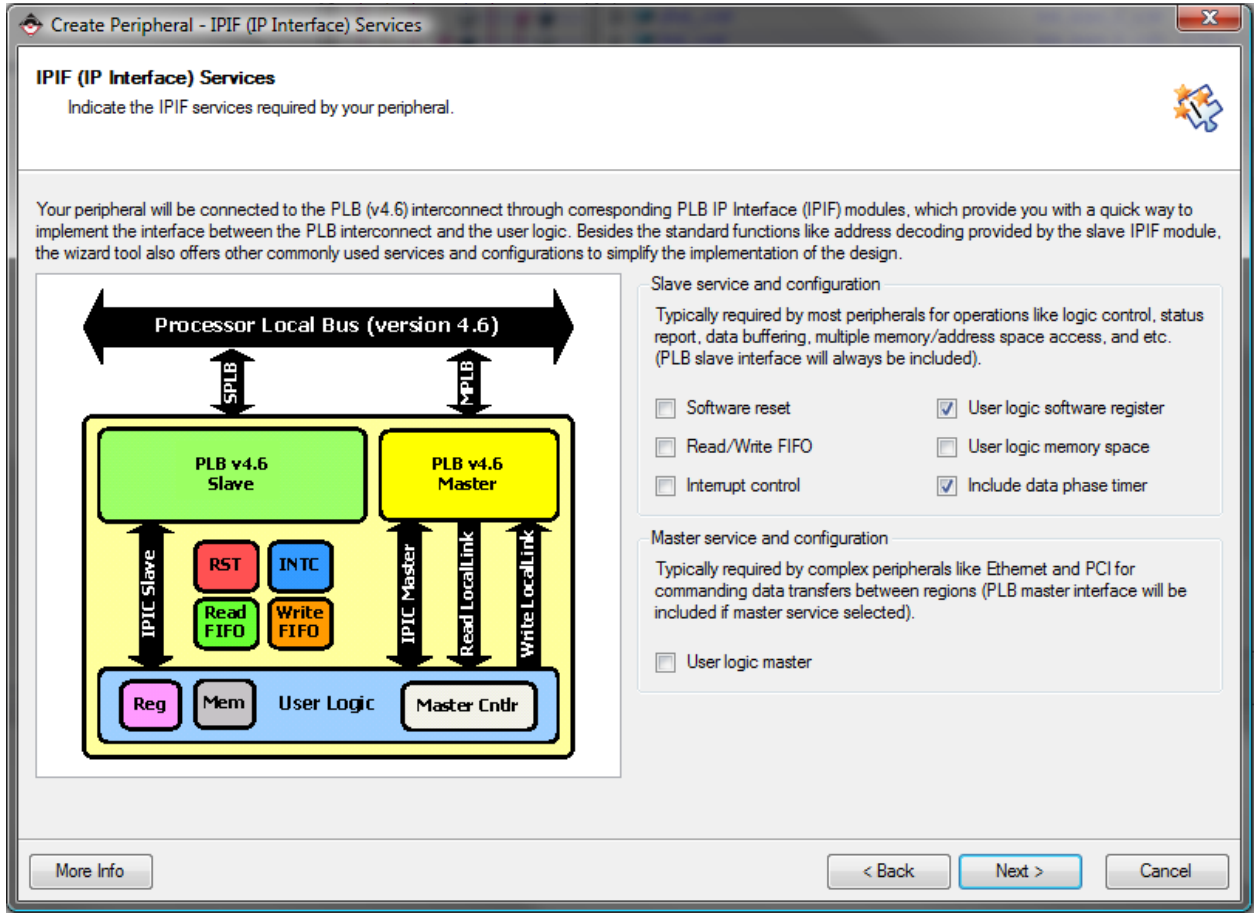


Figure 4.22: Peripheral Creation Window.

The next three sub sections go into detail on what the custom PLB slave peripherals include.

4.2.2.2 DVI IIC PLB Interface.

The IIC standard, developed by Philips, is a two-wire bidirectional serial bus that is used to provide communication between different low speed devices. This standard is capable of

arbitration between multiple masters and supports communication with multiple slaves and has a 7-bit addressing space for devices.

The DVI IIC interface uses a modified version of the IIC controller solution provided from www.opencores.org. This IIC controller has been redesigned to interface to a PLB bus as a slave. Several registers on the Chrontel DVI transmitter control need to be accessed to initialize the DVI transmitter. These registers access are done via IIC transfers over the DVI IIC bus. In particular, the DVI transmitter needs to be powered on and put into RGB bypass mode (analog mode). The controller is programmed to output 640x480 analog VGA. For more details on the configuration registers of the DVI controller please check the Chrontel CH7301c spec.

4.2.2.3 N64 PLB controller interface.

The N64 Controller, designed by Nintendo for use with the Nintendo 64 console, is used for manipulation of 2D and 3D objects in the FPGA based graphics system. It is mainly a tool used for debugging and was selected based on availability. Using it, objects on the screen can be selected, translated, scaled and rotated. The N64 Controller is shown below.



Figure 4.23: N64 Controller(20)

The N64 controller is a serial device with a three pins (Ground, 3.3V, and Control). The single wire control interface is a bi-directional open drain. A 500k weak pull down resistor is used on the control line. When a controller is not present on the line the pull down resistor drives the line low. When a controller is present a stronger pull up in the controller overrides the pull down and the value reads is high.

The PLB interface for the N64 controller has four registers used to control and read status from the device. These registers interface to a controller state machine which is responsible for both driving serial commands to the controller as well as reading and decoding serial commands from the controller. These registers are listed in Appendix B .

Software is responsible for polling the controller present bit in the status register to determine if the controller is indeed connected to the development board. If the controller is not connected the controller's state logic is pulled into reset. When the controller is detected, the controller state logic is pulled out of reset. It is then the software's job to periodically write to the button status or control word bit of the trigger register. Writing to either bit sends a serial pattern to the N64 controller which requests either the button status or the current controller word. After setting the trigger bit, software must poll the controller busy bit until it goes low indicating that the button status or control word has returned via a serial stream sent by the controller. Once the busy bit goes low, the button status or control word registers can be read to determine either which buttons has been pressed or to determine the current status of the controller. For this graphic system design, only the button status is ever needed as a means to manipulate objects on the screen.

4.2.2.4 Graphics Pipeline Registers PLB interface.

The processor needs a way to communicate with the graphics pipeline. Figure 4.24 shows how the graphics pipeline interfaces to the PLB bus.

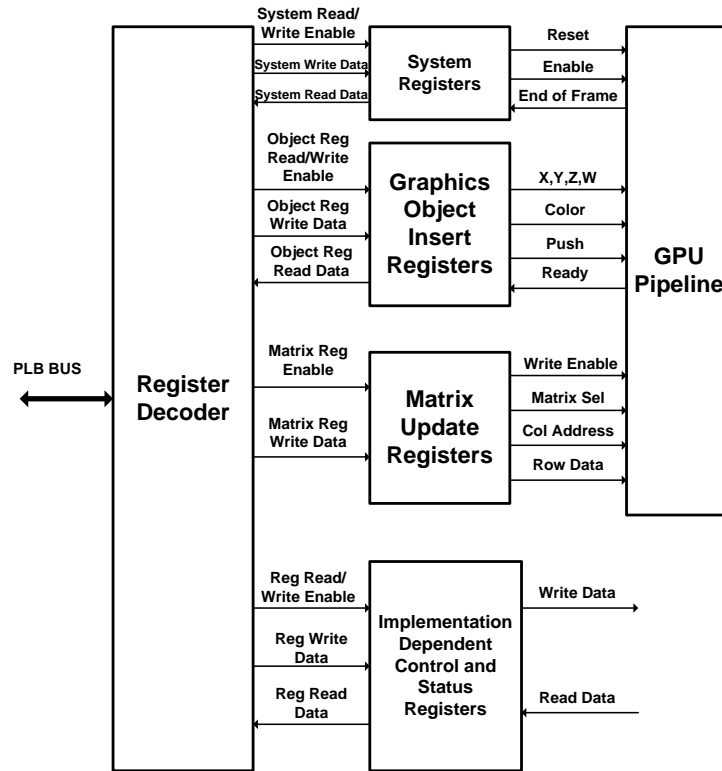


Figure 4.24: GPU Pipeline Register PLB Interface.

The graphics pipeline control register interface is logic specifically designed to interface to the PLB bus as a PLB slave device. This interface provides the needed communication between the Microblaze CPU and custom graphics pipeline logic over the PLB bus. In this block, various registers for configuration and debug of the graphics pipeline are present and can be seen in Appendix A. The graphics pipeline control register use the custom peripheral tool from EDK to create a PLB slave template. To implement the custom registers custom VHDL was integrated into the template

4.2.3 Graphics Pipeline Implementation

In this section the design of the graphics pipeline presented in Section 3.0 is implemented on the Xilinx Virtex 5 SX50 FPGA. Using the design from Figure 3.3 as a template, Figure 4.25 shows the entire pipeline.

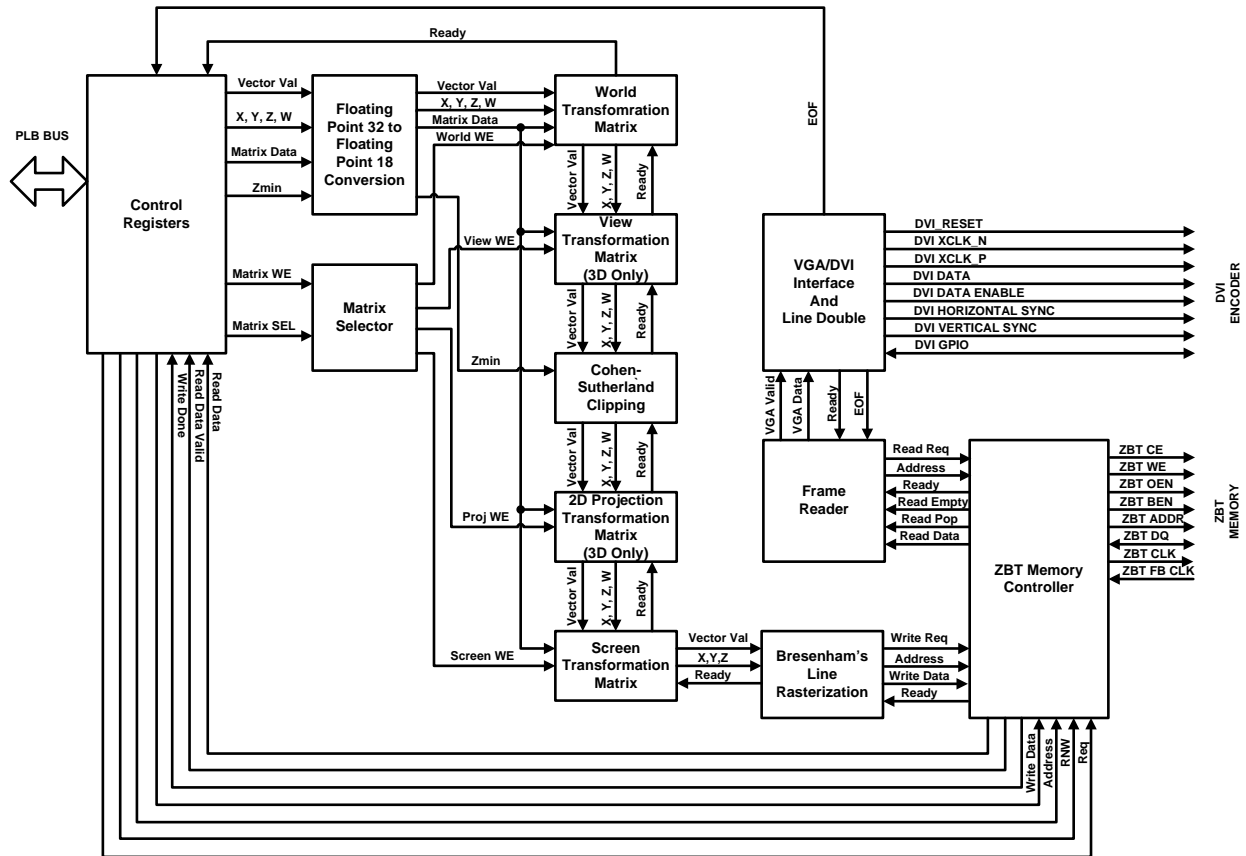


Figure 4.25: GPU Pipeline Top Level Implementation.

The graphics pipeline interfaces to the Microblaze processor via the processor local bus (PLB). The control register block acts as a PLB slave device. A PLB slave is a device which only receives transfer initiations from the PLB bus. The CPU acts as the PLB master and initiates all reads and writes to the graphic control registers. These control registers, defined in Appendix A, can be used to push the objects onto the pipeline as well as program the elements for the

transformation matrices. The pipeline then converts the objects from object coordinates to screen coordinates and rasterizes the objects on the screen. The world, view, projection and screen transformation matrix blocks all use the matrix multiplication accelerator design presented in Section 3.1.1. In addition, the Cohen-Sutherland clipping algorithm shown in Section 3.1.2 is used to clip graphics objects to the viewing volume. Externally, the graphics pipeline interfaces to a 256K x 36 ZBT SRAM. This memory is used as a frame buffer and stores the rasterized lines for each frame. The frame buffer uses double buffering. Due to this fact, the resolution is limited to 320x240. The frame reader reads the frame from the memory and drives the appropriate pixels out the DVI interface. The DVI interface executes line doubling to upscale the 320x240 image to 640x480 resolutions. Below detail about how each of the following blocks of the system was implemented is presented.

4.2.3.1 Floating Point Conversion and Matrix Selector

The CPU works with 32 bit single precision floating point numbers while the graphics pipeline uses 18 bit custom precision floating point numbers. Because of this fact, a conversion block is instantiated to do the conversion. This conversion block is created by Xilinx's Coregen IP generator discussed in Section 4.1. Using these conversion blocks the incoming point vectors, matrix elements, and the front clipping plane values are all converted by this block to 18 bit format. The floating point values destined for the world, view, projection and screen coordinate transformations are also converted to 18 bit format, but are multiplexed to the proper transformation matrix based on what registers are written in the GPU pipeline control registers.

4.2.3.2 Matrix Transformation and Selection

There are four matrix transformations in the 3D design and only two in the 2D design. Viewing and projection are not needed in 2D because the image is already 2D and can be displayed on the screen as is. Recall, the matrix multiplier design in Section 3.1.1.

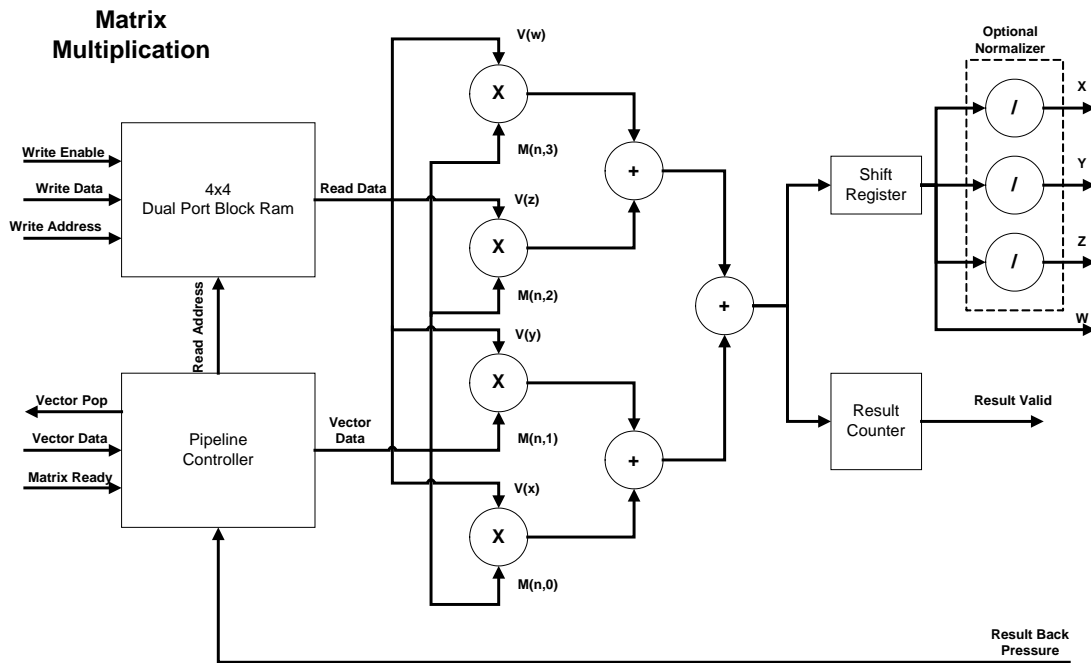


Figure 4.26: Matrix Multiplier Accelerator

This block is implemented using the 18 bit floating point multipliers, adders and dividers discussed in Section 4.1. The dual port block RAM is a 4 x 4 x 72 bit block RAM instantiated in the FPGA fabric. The block RAM is 72 bits wide because each entry contains an entire column of the matrix. Each point is stored in a 72 bit wide FIFO for which the pipeline controller of the matrix multiplier pops data. The pipeline controller cycles through all four columns stored in the dual port block RAM sending each row sequentially through the floating point multiplication and addition cores. The end result of the matrix multiplication is then shifted out. The optional normalizer block is only instantiated in the screen translation. This is so the final vector sent to the rasterization logic is normalized with respect to the w component (i.e. divided by w so that

w=1). This block was written completely in VHDL and the source code can be seen in Appendix C.3.

4.2.3.3 Cohen-Sutherland Clipping.

The clipping stage trims lines around the defined clipping volume. Recall the Cohen-Sutherland Clipping design presented in Section 3.1.2 and shown again in Figure 4.26.

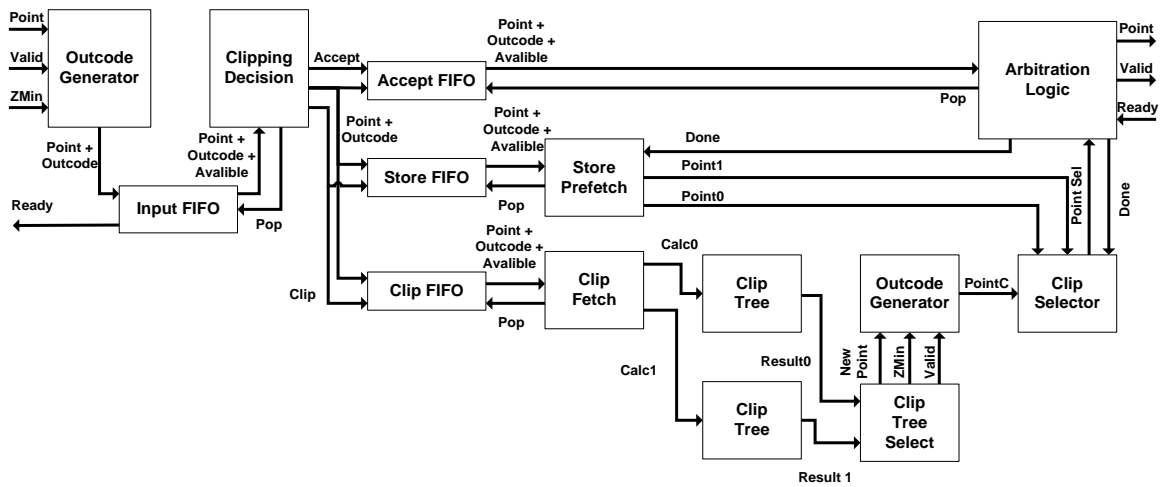


Figure 4.27: Cohen Sutherland Clipping Implementation

This block is written completely in custom wrote VHDL with a few acceptations. The outcode generators utilize 18 bit floating point compare IP cores to determine each points outcode. The other exception is that the Clip Tree which calculates the plane intersections is composed of 18 bit floating point multipliers, adders and dividers. The source code for the custom VHDL is located in Appendix C.7.

4.2.3.4 Bresenham's Line Rasterizer

The rasterization converts the graphics objects to physical pixels on the screen. Recall the Bresenham's Line Rasterizer design presented in Section 3.1.3 and shown again in Figure 4.28.

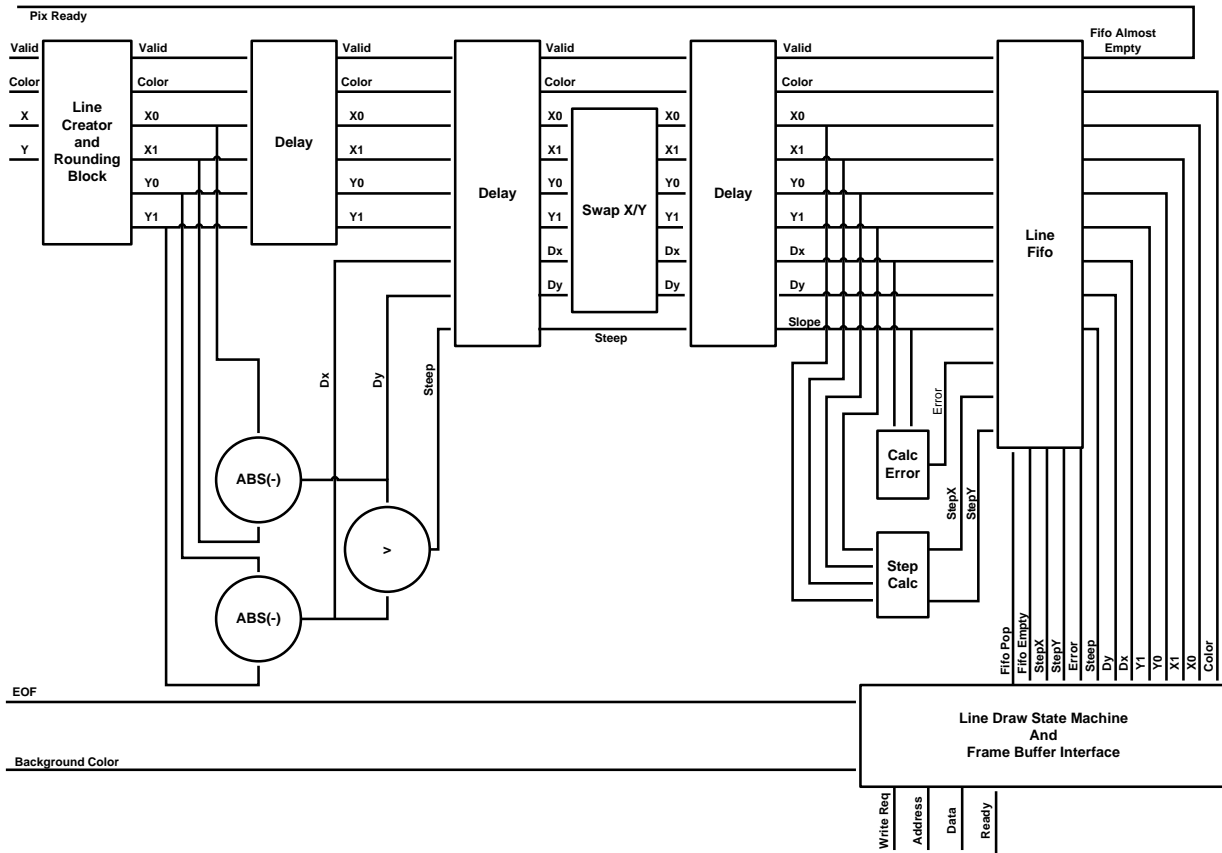


Figure 4.28: Bresenham's Line Rasterizer Design

This block is written completely in custom wrote VHDL as it is completely integer based and requires no floating point cores. The source code can be found in Appendix C.9.

4.2.3.5 Frame Buffer and the ZBT Memory Controller

The frame buffer uses double buffering to prevent artifacts from occurring on the screen. Artifacts occur because when a single buffer is used, the line rasterizer and display interface can be writing and reading the same locations from memory in some corner case conditions. Figure 4.29 shows the design of the frame buffer and memory controller. In actual implementation the memory is the ZBT SRAM provided by the ML506 development board.

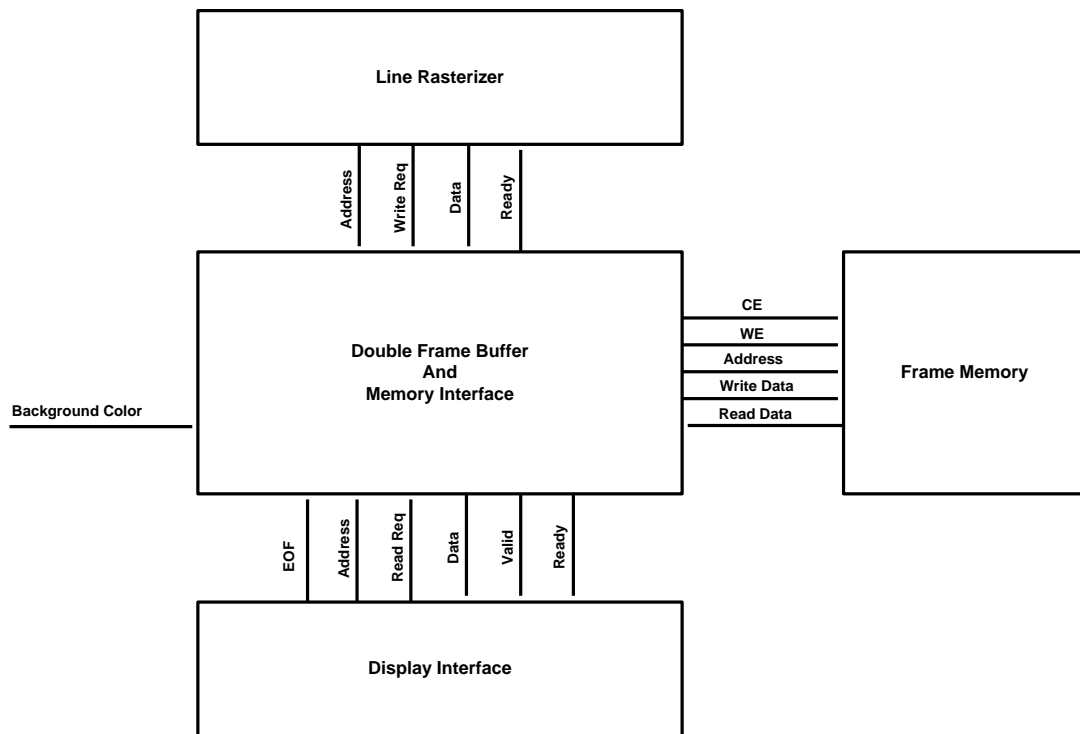


Figure 4.29: Frame Buffer Interface with Frame Memory

Both the frame buffer and memory controller are written in VHDL and can be seen in Appendix C.10.

4.2.3.6 VGA Display Interface and the Line Doubler.

The most common display interface used in computer graphics today is VGA. On a VGA interface there are three signals (red, green, blue) that send color information to a VGA

monitor. These signals each drive an electron gun that emits electrons which paint one primary color at a point on the monitor. The signal varies between 0 and 0.7 V which in turn controls the intensity of each color component. The three colors together combine to form a pixel. An image (or frame) on a monitor is composed of h lines each containing w pixels. A full frame is expressed as a $w \times h$ as its size. In this example $w = 640$ and $h = 480$. In order to draw a frame the deflection circuits in a monitor move the beams of electrons from left to right and top to bottom of the screen. To control the deflection circuit's two pulses (horizontal and vertical sync) must be generated. Below are typical timing diagrams for horizontal sync and vertical sync.

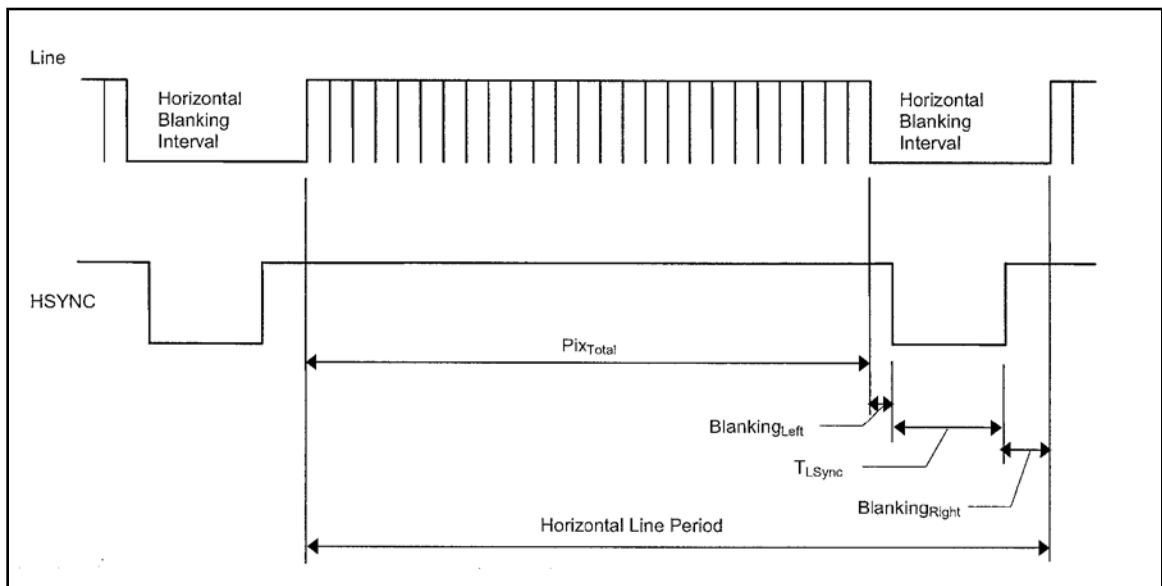


Figure 4.30: VGA Horizontal Sync Timing.

For a 640x480 @ 60Hz VGA display the timing parameters or as shown below:

Table 4.1: VGA Horizontal Timing Table.

| H Sync Timing Param. | Period | Number of Clock Cycles at 25Mhz. |
|----------------------|----------------------|----------------------------------|
| T_{Lsync} | 3.84usec | 96 |
| $Blanking_{Left}$ | 0.64usec | 16 |
| $Blanking_{Right}$ | 1.92usec | 48 |
| Pix_{Total} | 25.6usec | 640 |
| Horizontal Freq | 32.0usec or 31.46Khz | 800 |

The vertical sync pulse is enabled and clocked off the horizontal sync pulse. The timing information is shown below.

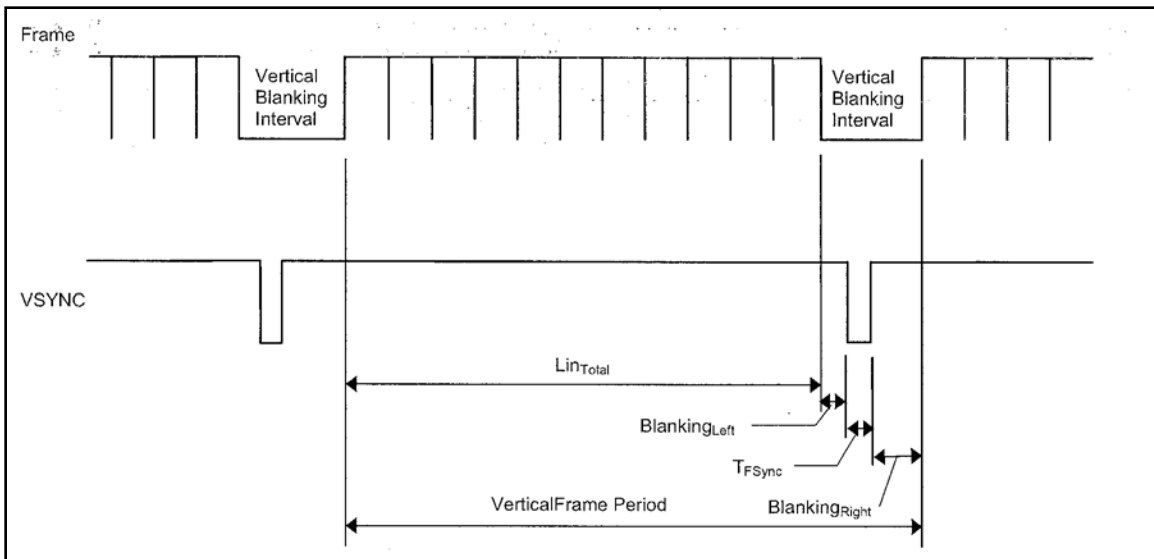


Figure 4.31: VGA Vertical Sync Timing

For a 640x480 @ 60Hz VGA display the timing parameters are as shown below:

Table 4.2: VGA Vertical Timing Table

| V Sync Timing Param. | Period | Number of Horizontal Sync pulses |
|----------------------|-------------------|----------------------------------|
| T_{Lsync} | 64.0usec | 2 |
| $Blanking_{Left}$ | 352.0usec | 11 |
| $Blanking_{Right}$ | 992.0usec | 31 |
| Pix_{Total} | 15.36msec | 480 |
| Vertical Freq | 16.79msec or 60hz | 524 |

The output display is running at 640x480 @ 60Hz but the internal GPU pipeline logic only supports 320x240 @ 60Hz. A line doubling circuit written in VHDL bridges the gap between the 640x480 and 320x240 resolutions. In simple terms, the line doubling logic sends out two identical horizontal lines to the display for every one horizontal line within the GPU pipeline. The same two to one ratio holds true for the vertical lines. This is accomplished by instantiating a small line buffer which as each odd horizontal line is received from the GPU pipeline the line is also stored in the line buffer. On the even horizontal lines, the data is read from the line buffer sending the horizontal line stored in the line buffer. In addition, as each pixel comes into the line doubler, the pixel is sent out twice effectively reducing the vertical resolution by half. Using this logic, the GPU pipeline can run at the low resolution of 320x240 while the physical display thinks it's displaying a 640x480 image. The source code for the line doubler can be found in Appendix C.18.

4.3 GRAPHICS PIPELINE FUNCTIONAL TESTBENCH

Now that the GPU is completely implemented, the design can be tested to verify that the graphics processor functions as specified. To accomplish this, a VHDL testbench was developed with the purpose of simulating the design using Mentor Graphic's Modelsim. Modelsim is a digital circuit simulator that aids in design and verification of digital systems. Modelsim provides the benefit of allowing the tester to probe internal digital signals that would otherwise be unreachable in conventional testing. This gives the tester more visibility into the design.

While in theory the Microblaze processor could be simulated for functional correctness. Given that it is a Xilinx provided intellectual property (IP), it is taken for granted that it works properly. The graphics pipeline is the custom logic in the design and hence will be the component tested in simulation. The testbench is designed to mimic the functionality of the CPU. Below is a very high level view of a functional testbench developed for the graphics pipeline.

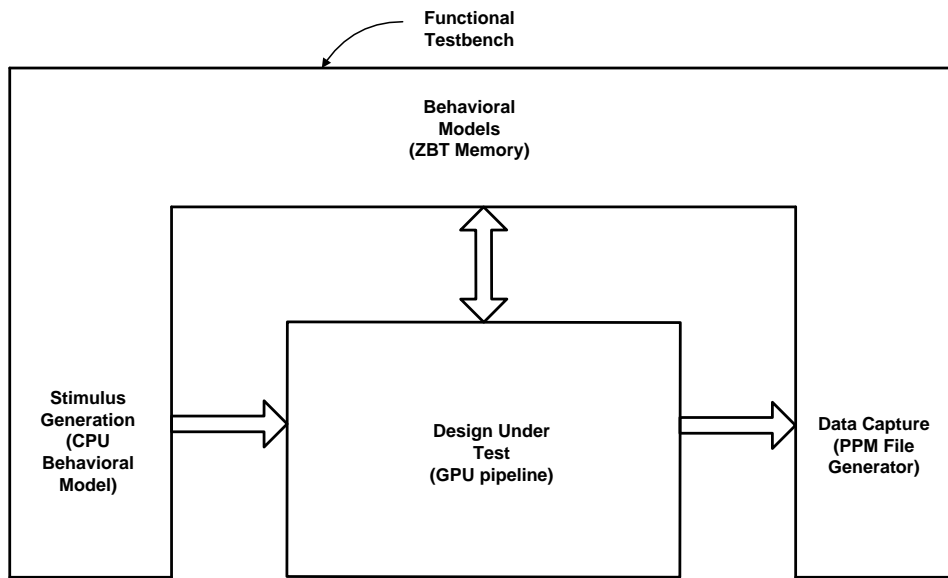


Figure 4.32: GPU Testbench Block Diagram

The stimulus generator acts as a behavioral model for the CPU as well as other functions such as clock and reset generation. In addition, the ZBT frame buffering memory is mimicked using a behavioral hardware description language (hdl) model provided by Cyprus. This model is cycle for cycle accurate and provides confidence that the logic interfacing to the ZBT memory functions properly. On the other end of the testbench the data capture block interoperates the outgoing DVI signals and generates an image file of the frame. In the testbench itself, the GPU's graphics pipeline is instantiated so it can be interfaced to by the stimulus generator, data capture and the ZBT memory behavioral model. Using this setup, stimulus configuration files can be

used to push a variety of objects through the graphics pipeline. Modelsim does not support IEEE754 floating point data-types for debug in its waveform tool let alone the custom 18-bit floating point bit values used in this design. This makes debugging wave forms quite complicated. To get around this problem, a custom VHDL package was created to convert both custom 18-bit floating point bit values and 32-bit IEEE754 floating point bit values to decimal values to be displayed on the simulators console. In addition, the image file generator can be used to actually see what the output to a monitor would look like.

Each specific test must handle the following operations within the testbench.

- Handles all the initialization needed by the pipeline before objects are loaded into the pipeline.
- Loads the simulation files with various graphics objects to be pushed into the pipeline for each frame.
- Inputs the values for the translation matrices as needed by the system. The screen and projection translation matrix are loaded only once, the viewing translation matrix is loaded for each frame and the world translation matrix is loaded once for each object.
- Handles polling for the end of frame interrupt which is used to determine when each frame ends and begins.

For each frame, the data capture interprets the outgoing DVI interface and converts the information to a PPM frame. A PPM files stands for a portable pixmap file where raw RGB data is stored from left to right and top to bottom fashion. PPM files are convenient because DVI/VGA displays output RGB data in left to right top to bottom format. PPM files store data the same way, from left to right and top to bottom. Also, the header for the file is minimal

making the file almost completely raw RGB data. In addition to the PPM files, the testbench displays results which give information on the test results on the Modelsim console.

This section goes through a step by step process of how data is progressed through the pipeline. First the pipeline waits for an end of frame interrupt from the VGA controller. After the interrupt occurs, the viewing and world transformation matrix is updated using the matrix mailbox interface. The waveform in Figure 4.33 shows the world transformation matrix being programmed.

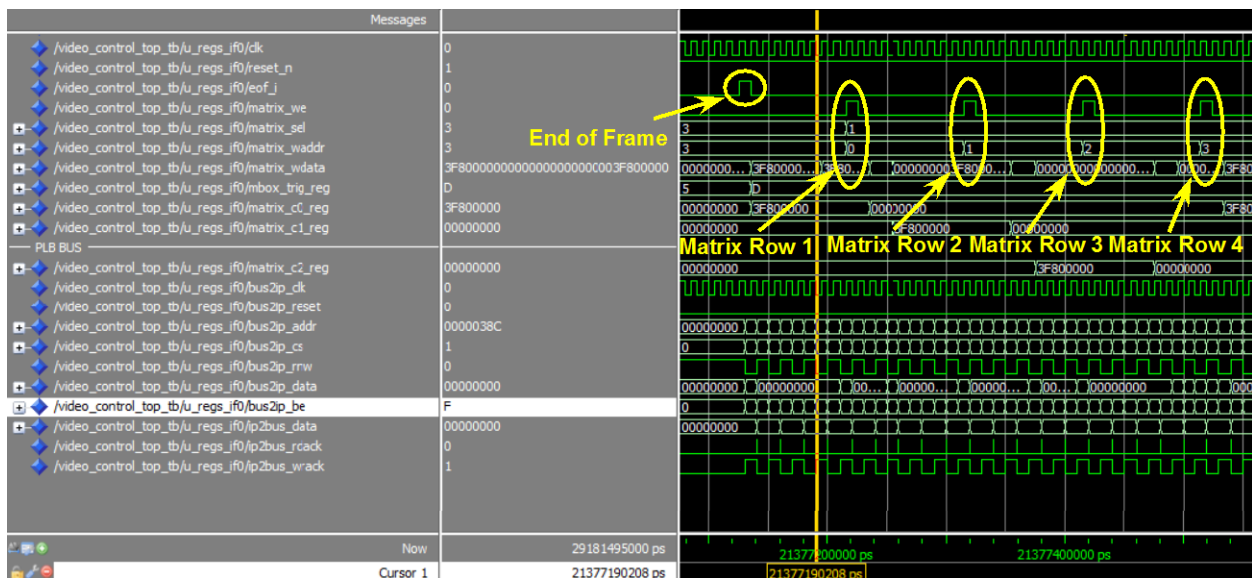


Figure 4.33: Matrix Programming in Simulation

Once the transformation matrices are programmed objects can begin being pushed into the graphics pipeline after the VGA's end of frame signal. It is here that the individual (x,y,z,w and color) vectors are pushed into the pipeline as shown in the wave in Figure 4.34.

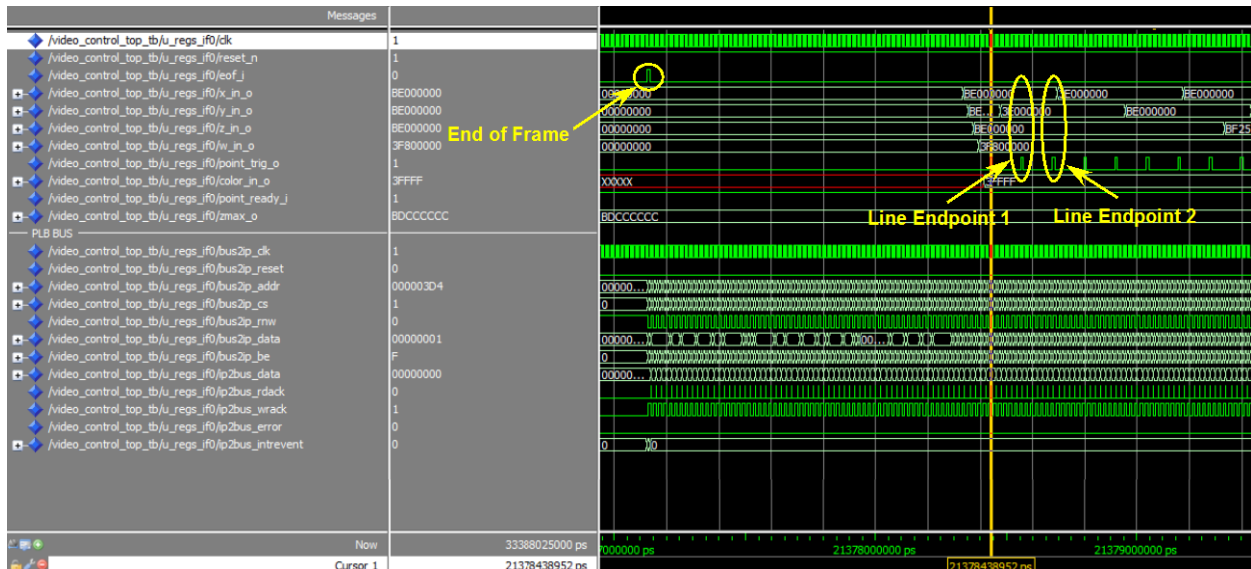


Figure 4.34: Pushing Line in Simulation

The object data is then passed through the various matrix multiplication blocks to transform between the various coordinate spaces. The pushing of object data through the world translation multiplication and the multiplication's results are shown in Figure 4.35.

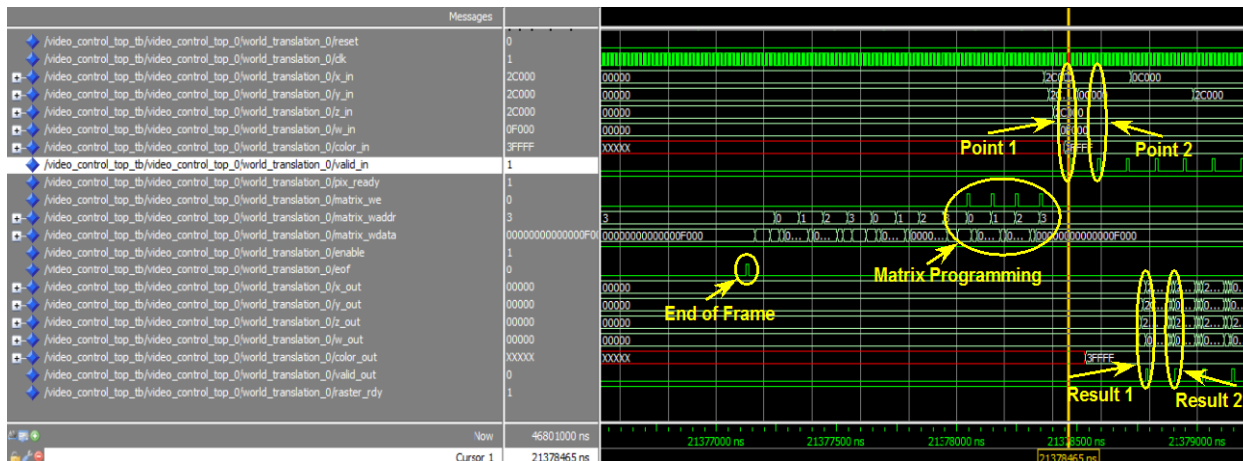


Figure 4.35: Matrix Multiplication in Simulation

The object data goes through each multiplication stage of the pipeline. From there the objects are clipped, rasterized and stored in the frame buffer. On the next frame, the objects are read from the buffer and driven out on the DVI interface.

The data destined to the DVI interface is attached to a PPM file generator. Here a PPM frame is generated for each actual frame driven out on the display interface. An example of a unit cube PPM file generated by the testbench is shown below.

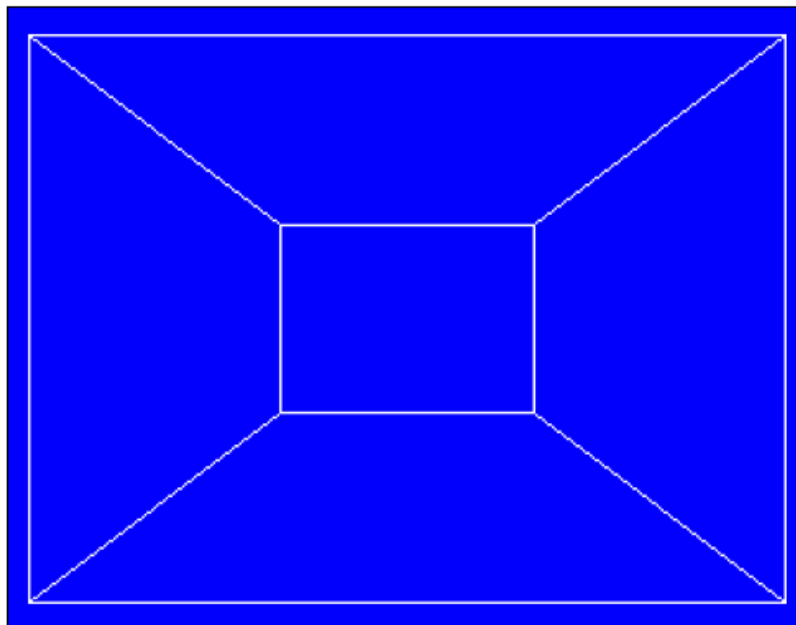


Figure 4.36: PPM unit cube.

Pushing various test images and verifying them using the waveform and their generated PPM frame, establishes that the basic functionality of the GPU's graphics pipeline was functioning properly. Hence it was safe to implement the GPU into fully synthesized logic using the FPGA. By synthesizing the logic, an idea of the resource utilization needed by the GPU can be attained and further functional and performance testing can be performed.

4.4 GPU SYNTHESIS

The following section goes into the details involved with FPGA synthesis of the GPU design on the Xilinx Virtex 5 SXT 50 device. Synthesis here involves both the graphics pipeline as well as the Xilinx Microblaze CPU. In addition, logic is needed to synthesize any external interface peripheral controllers such as the DVI interface, ZBT memory controller, and N64 controller interface. In this section first the steps involved in design synthesis are presented followed by the actual synthesis resource utilization results.

4.4.1 Xilinx EDK and Microblaze

The CPU is used for various tasks within the design. Using the processor, C programs were written which calculate the elements of the matrix transformation, initialize the GPU, push graphics objects onto the GPU pipeline, and manipulate graphics objects in software. The CPU also has various peripherals which interface to external devices such as the DDR2 SDRAM controller, DVI controller, N64 controller, and serial UART. The UART is used to display information and results from any application run on the CPU. All compiled software executables are stored in external DDR2 SDRAM. The Microblaze CPU, DDR2 SDRAM controller, and serial UART are all intellectual property provided by Xilinx's Embedded Development Kit (EDK) software. The DVI controller is a modified design provided by opencores and the N64 controller is a custom built peripheral meant for testing. The figure below shows all these peripherals connected to the PLB in EDK.



Figure 4.37: Xilinx’s EDK showing processor sub-system

Xilinx's EDK handles the creation of the Microblaze CPU system. It also handles the interconnection of peripherals such as the UART and DDR2 SDRAM memory controller via the PLB bus. In addition, the graphics pipelines control registers are mapped to a specific location within the PLB buses addressing space. EDK also handles the compilation and syntax checking of executable code written for the Microblaze processor. The compiled executable is stored in and executed from DDR2 SDRAM.

4.4.2 ISE and Full GPU Synthesis

The Microblaze system is just a portion of the design hierarchy. The Microblaze subsystem is then added to the GPU design as shown in the figure below. This figure shows the hierarchy used by ISE to synthesis, map, and place and route the FPGA design.



Figure 4.38: ISE 3D GPU hierarchy

The figure above shows the Microblaze processor instantiated within the system and name u_cpu0. The other major block is the GPU which is named u_gpu0. The GPU consists of the four translation matrix blocks, clipping logic, rasterization logic, VGA/DVI interface and the ZBT frame buffer interface. In addition, 'gpu.ucf' is a constraint file which holds pin placement constraints for the external peripherals on the Xilinx Virtex 5 SX50 development board. The 'gpu.ucf' file also hold timing constraints directing the synthesis to meet the desired timing requirements of 100MHz.

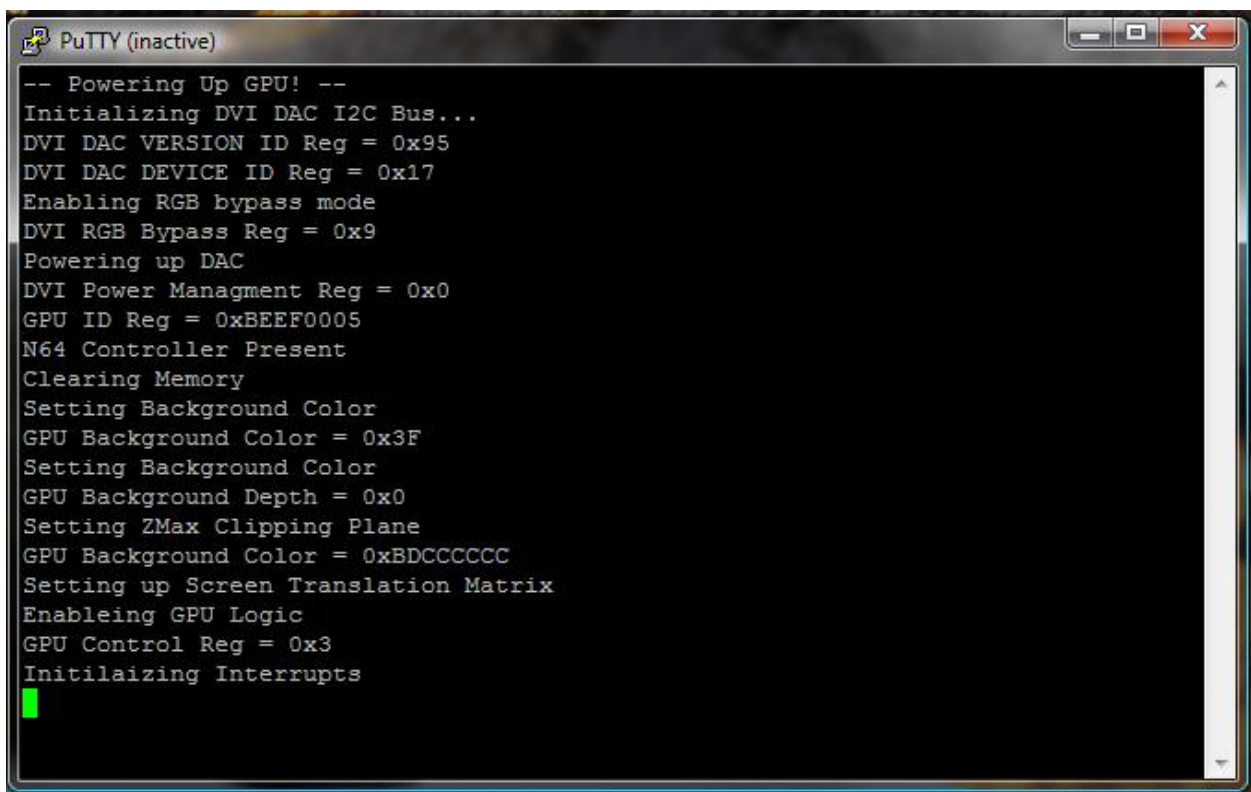
Xilinx's ISE tool is used for synthesis, place and route, and program file generation. After these steps are completed, the programming file is loaded onto the FPGA's configuration prom. Upon power up the FPGA reads the data from the non-volatile configuration PROM and configures it's the FPGA's internal fabric accordingly.

4.4.3 Synthesis Results

The full 3D GPU system uses 18,860 or 57% of the available LUTs and 23,630 or 72% of the available flip flops in the Xilinx Virtex 5 SX50 device. Moreover, 2,016 KB or 42% of the Virtex 5's block memory is used with the 3D GPU. This leaves a substantial amount of area that could be used for other graphical functions (such as lighting, texturing or shading). This shows that a basic 3D GPU is feasible in today's modern FPGA devices. In fact, for this particular design a smaller, less expensive Virtex 4 or Spartan 3 may be able to be used. Now that the design is put into hardware, it can be verified to work in actual hardware by using test software application which runs on the Microblaze CPU.

4.5 SOFTWARE BASED HARDWARE TESTING

The hardware was tested using a C application designed to be run on the Microblaze processor. The test software performs several functions. The first is initializing the system. System initialization involves configuring the DVI controller via its IIC interface, enabling the N64 control interface, configuring the clipping logic and setting the element values of the screen translation matrix. All of these steps can be seen below in Figure 4.39.

A screenshot of a PuTTY terminal window titled "PuTTY (inactive)". The window displays the following text:

```
-- Powering Up GPU! --  
Initializing DVI DAC I2C Bus...  
DVI DAC VERSION ID Reg = 0x95  
DVI DAC DEVICE ID Reg = 0x17  
Enabling RGB bypass mode  
DVI RGB Bypass Reg = 0x9  
Powering up DAC  
DVI Power Managment Reg = 0x0  
GPU ID Reg = 0xBEEF0005  
N64 Controller Present  
Clearing Memory  
Setting Background Color  
GPU Background Color = 0x3F  
Setting Background Color  
GPU Background Depth = 0x0  
Setting ZMax Clipping Plane  
GPU Background Color = 0xBDCCCCC  
Setting up Screen Translation Matrix  
Enableing GPU Logic  
GPU Control Reg = 0x3  
Initilaizing Interrupts  
█
```

Figure 4.39: Serial terminal output from test software.

The figure above shows the initialization output of the FPGA's Microblaze processor. This window is a console program running on a PC.

After the GPU is initialized the test software begins pushing graphics objects to the GPU pipeline. At the same time, the processor is checking the status of the N64 controller. The controller is used to move the objects on the screen as well as change the camera angle of the 3D

scene. A sample screen where the processor is pushing a single cube and single pyramid into the graphics processor is shown in Figure 4.40.

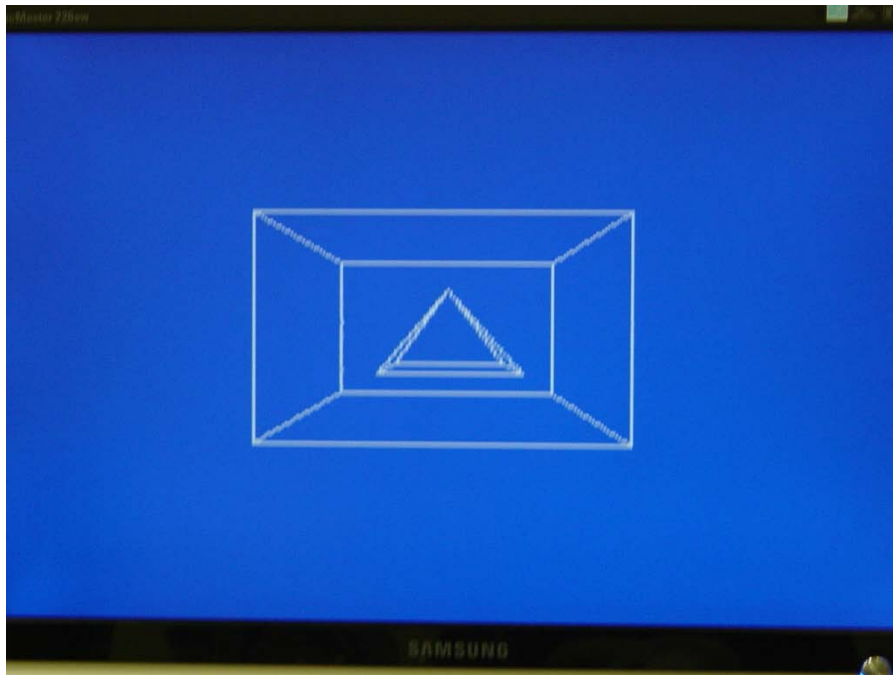


Figure 4.40: 3D Graphics Processor Output

Several features needed to be tested in real hardware. The following images show examples of the same objects in Figure 4.40 with rotation, scaling, translation and clipping applied. Figure 4.41 shows the pyramid and cube rotated about the z-axis.

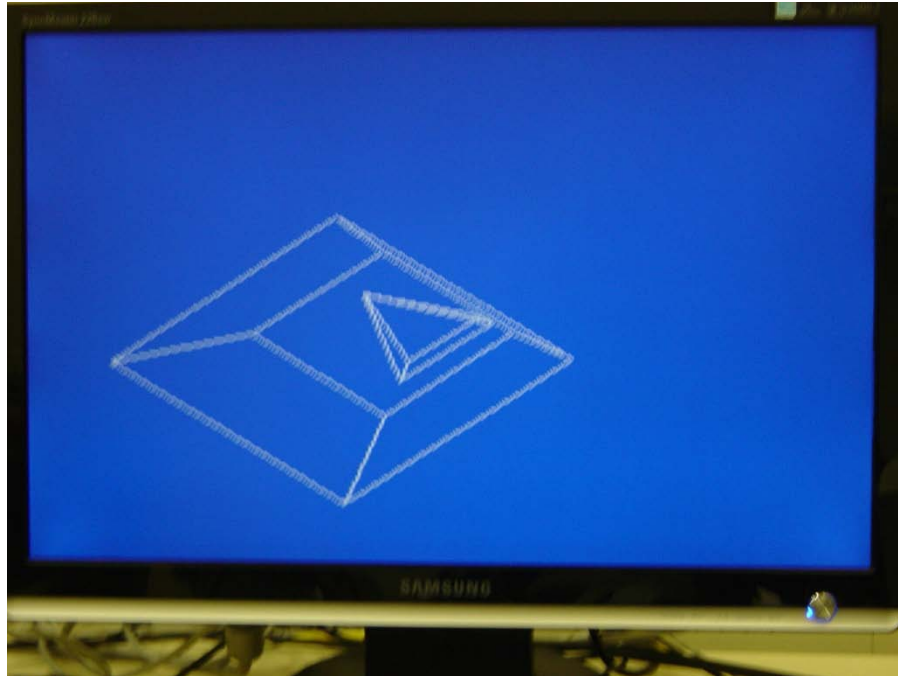


Figure 4.41: 3D Object rotation.

In Figure 4.42 the rotate object is then reduced in size using scaling.

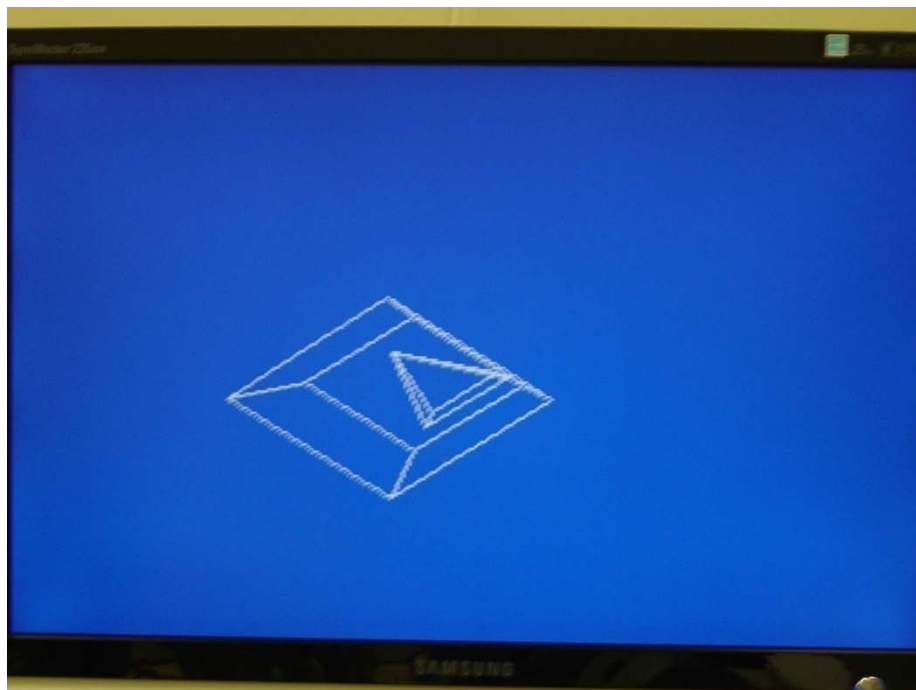


Figure 4.42: 3D Object Scaling

Lastly clipping and translation is tested in Figure 4.43 by translating the object off screen and verifying that the objects are clipped properly.



Figure 4.43: 3D Objects Translated and clipped.

Using this basic test, it can be seen that the basic functionality of the system is working. Next is to try a more complex model to more exhaustively test the system.

A more complex model is rendered in Figure 4.44. This model is a plane containing over 5000 polygons. All of this is rendered at 60 Frames per second.

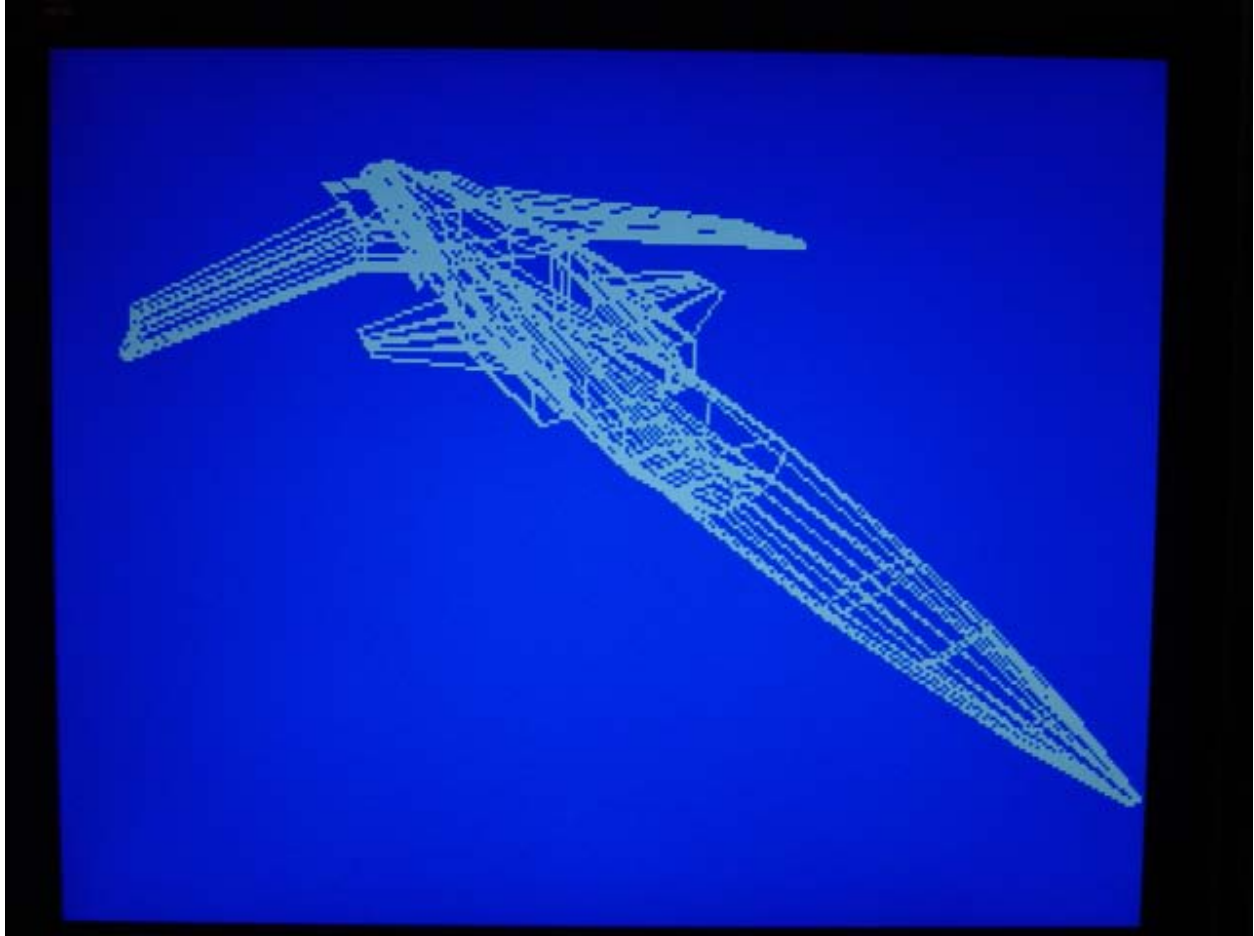


Figure 4.44: Model Plane Rendered using GPU

A picture of the entire hardware system which was used to render the above objects is shown in Figure 4.45.



Figure 4.45: Development Kit with JTAG, VGA and N64 Hardware

This software test code proves that the design is functionally working as expected. The test code is located in Appendix D.

5.0 SUMMARY, CONCLUSIONS, AND FUTURE WORK

5.1 SUMMARY AND CONCLUSIONS

This thesis set out to demonstrate that it is feasible for both 2D and 3D real time computer graphics processing units to be implemented in modern FPGA devices. In particular, this thesis set out to implement a wireframe based graphics engine capable of rendering objects in a 2D or 3D virtual world. The requirements also asked for implementation of a free roaming viewpoint or camera so that a 3D virtual world can be viewed from any perspective. Lastly, the graphics processing unit must be able to interface with today's standard display technologies. Accomplishing these objectives would establish FPGA's as viable graphic processing elements in embedded systems.

To meet these goals, some extensive research was initially performed on implementation of graphics processing. Typical graphic pipeline algorithms that can be implemented in an FPGA, such as Affine Geometric Transforms, Bresenham's Line Algorithm for rasterization and Cohen-Sutherland's clipping algorithm were highlighted. Each algorithm was analyzed to see if hardware acceleration could be efficiently implemented. Bresenham's Line algorithm and Cohen Sutherland's clipping algorithm were designed completely in hardware using floating point cores provided by Xilinx along with custom designed logic. In addition, Affine Geometric Transformations involve implementing matrix multiplications in

hardware. These transformations require Xilinx's floating point cores and some custom logic design. These transformations also require software assistance for matrix element calculation.

The software used to control the system required a central control processor. The Xilinx Microblaze processor was added to the design and provided a means for storing the graphics objects, offloading some geometric transformation element calculations, and system initialization.

To implement this design, a Virtex 5 SX50 ML506 development board was used. The ML506 has the necessary display interfaces, memory, and logic resource to guarantee that development hardware did not present any bottleneck during design. Using this development board, the GPU implementation used a combination of a hardware accelerated graphics pipeline with some processor assistance. This yielded a design with reasonable logic utilization and good performance.

Before synthesis of the system, initial verification was done using a VHDL testbench to verify that the GPU was functionally correct. Upon initial confirmation, the actual hardware was synthesized on the ML506 development board. Maximum performance met the design performance requirements with the ability for the geometry engine to process 8.33 million polygons per second and the rasterizer to fill 100,000,000 pixels per second ignoring processor overhead. The design took up a reasonable 60-70% of the device leaving more room for some of the features discussed in Section 5.2 as well as any addition embedded system and system on chip features discussed in Section 1.2.

5.2 FUTURE WORK

The design and implementation provides a good starting point for using FPGA devices as for graphics processing applications. Given this fact, additional features could be added to the FPGA solution presented in this thesis. In addition, some design optimizations can be made to make the current design function even more efficiently. This section details what feature additions are necessary to implement a more complete and efficient graphics hardware accelerator and justify how it is viable in today's FPGA. In addition, it touches on some advanced features such as partial reconfigurability.

5.2.1 Feature Additions

The current graphics accelerator designed in this thesis uses only wireframe objects for rendering. Wireframes draw only the perimeter of an object as illustrated back in Figure 1.3. For more realistic looking objects, functions such as texturing, illumination, shading, hidden surface removal and anti-aliasing are needed.

Texturing or texture mapping is a method for adding surface detail to the faces of 3D rendered object. Texturing applies what is known as a texture map to the surface of a polygon. Textures are formed from a standard two dimensional image where each pixel within the map is known as a texel. This texture map is defined on its own coordinate system (u,v) . This process is similar to applying wrapping paper around a plain box. This process requires additional texture storage memory as well as additional matrix transformations to map the text coordinates to pixel coordinates.

Illumination is another common feature in graphics processing that adds the sense of realistic lighting and shading. Various illumination models exist from ambient light models, to single and multiple point light sources. Shading is the process of applying a given illumination model to every visible point within the scene. Shading and lighting involve the calculation of surface normals to polygons, which can be calculated using matrix math for which the hardware is already designed for coordinate transformations.

In most graphics applications hidden surface removal is done using the z-buffering technique. Z-buffering is the management of depth coordinates in a 3D object. This can often be done in the frame buffer by storing a corresponding z coordinate with each pixel. In fact, this design divided the ZBT frame buffer memory words into 18 bit segments for this purpose. The first 18 bits store the RGB color of the pixel and the second 18 bits store the floating point value of the current z coordinate of this pixel. Z buffering is quite simple, for each pixel that is edited in the frame buffer the z value is read. If the new pixel's z value falls in front of the old pixels z value then the pixel's z value and RGB value are overwritten. Otherwise, the pixel is not written to the frame buffer. This method could easily be added to this thesis's design but is of minimal use in wireframe designs and due to time constraints was not implemented. Also using z buffering doubles the frame buffer's memory bandwidth because each pixel in the frame buffer must be initially read and have the old z value compared with the new z value. If the pixel is visible, it is then written to the frame buffer. Section 5.2.5 explores with ways to improve rasterization and frame buffer performance.

5.2.2 Direct Memory Access

The graphics accelerator designed in this thesis is a fully functional wireframe graphics renderer. This subsection highlights various design optimizations and changes that could be made to improve performance of the design.

Section 4.5 describes how software is responsible for pushing all the graphics objects onto the graphics pipeline for every frame. In the software test application, graphics objects are stored in DRAM. Upon every end of frame interrupt, the processor must handle fetching the objects from memory and writing these objects over the PLB bus. The PLB writes are targeted to the graphic's pipelines control registers in order to push these objects onto the input FIFO of the graphics pipeline. This is a very inefficient requiring the processor to perform many costly writes for every frame. This overhead could be eliminated by first reserving a section of DRAM for object storage and reserving another section for software code. Figure 5.1 shows the purposed memory division.

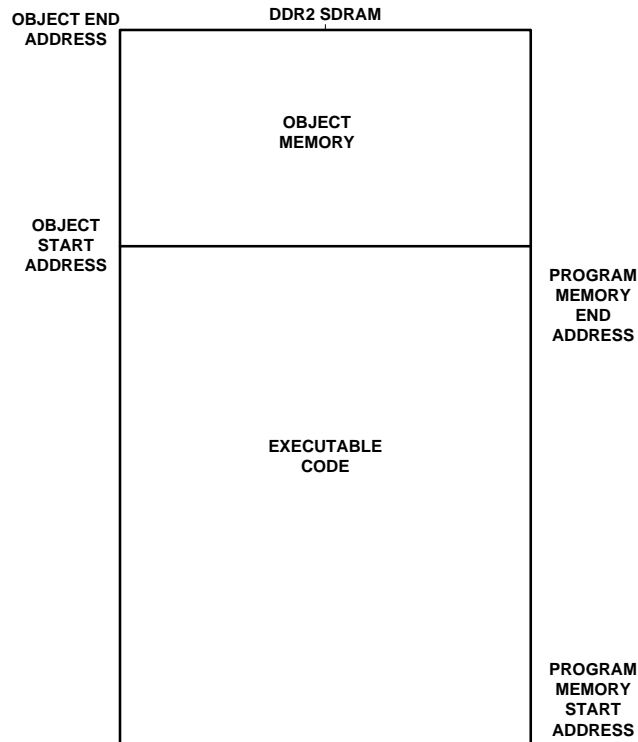


Figure 5.1: Program Memory and Object Memory Division

By dividing the memory, the processors function can be mitigated to only updating or deleting objects within the object memory. Direct memory access can be used by the graphics pipeline to fetch the object needed for each frame. Figure 5.2 shows a proposed design involving a multiport memory access to the DDR2 SDRAM. Xilinx provides a Multi-port Memory Controller (MPMC) IP core that connects directly to the PLB bus. This interface provides direct read/write access to the DDR2 SDRAM and handles the arbitration between ports. It is still the responsibility of the CPU, upon receiving an end of frame, to update the object memory and then to indicate to the graphics pipeline that all updating for the current 2D or 3D frame has been completed for this frame. When the frame is complete the CPU sends a Start DMA signal in Figure 5.2 to indicate for the pipeline to start reading. Since the graphics pipeline handles reading the objects from object memory this leaves the general purpose CPU more cycles to do other tasks.

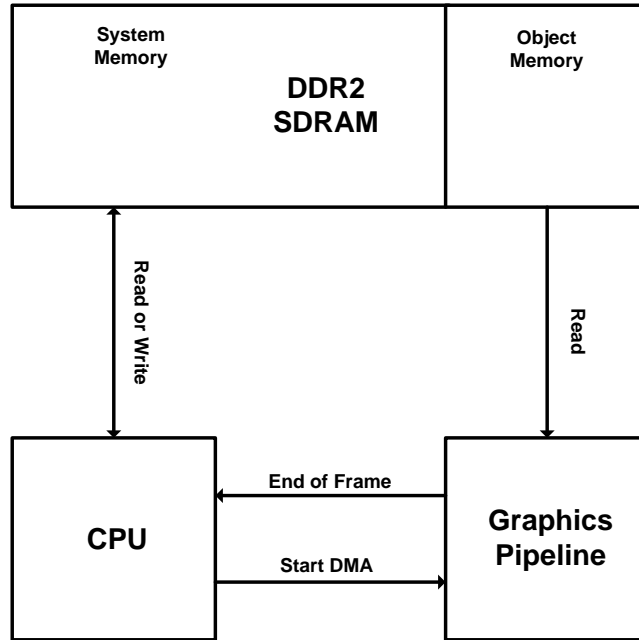


Figure 5.2: Graphics Pipeline DMA

5.2.3 Transformation Element Calculations

Another core operation of the CPU is calculating the matrix elements for each transformation matrix. In the current design this is done in software for convenience. Recall the various transformation matrices from Section 2.2. These matrices require cosine and sine trigonometric functions as well as additional floating point multiplication, additions and division for calculation of each matrix elements. Moreover, each transformation matrix requires different calculations for each of its elements. This requires special purpose hardware for each matrix. Use of a software solution for viewing, projection and screen coordinates is not costly because the screen transformation is only updated at power up and the viewing projection transformation matrices are only updated once per frame. Since the frame rate of a monitor is 60Hz, this means the viewing and projection matrices only needs updated once every $1/60^{\text{th}}$ of a second. Using a

100 MHz processor with a period of 10ns, the time needed to do these calculations will be negligible.

The world coordinate transformation must be updated once per object, this can happen in a non deterministic number of times within one frame. As the design stands now the software calculates each entry of this matrix. This transformation requires floating point cosine, sine, multiplication and addition functions. Performing these calculations for each object causes a huge bottleneck for the system as the graphics pipeline must wait for the new world transformation matrix to be updated before it can process any new graphics objects. To alleviate this problem, floating point cosine and sine hardware acceleration must be implemented as well as using addition multiplication and addition floating point cores define in Section 4.2.1. Implementing these operations will take up more FPGA area, but will allow the graphics pipeline to stream graphics objects without being interrupted by the processor.

5.2.4 Using external processor over PCI express.

As more complicated graphics features are needed, FPGA real estate will be at a premium. Fortunately, Xilinx Virtex 5 FPGAs have an integrated PCI Express block which are compliant with Specification 1.1. x1, x4, and x8 lane speeds are supported. The PCI express could be used to eliminate the need for an on board processor. In Microblaze systems on the Virtex 5 SX50, this could save roughly 30% of the chips resources. This additional logic could be used to implement more advanced functions such as texturing or illumination. Using PCI express enables communication with an external processor. An external processor may be

significantly more powerful than a Microblaze, potentially adding better performance to the overall system.

5.2.5 More Parallelism in Rasterization

Rasterization has the potential to become the major bottleneck for the system. The geometry engine of the graphics pipeline can process a triangle every 12 cycles, while rasterization of a wireframe triangle is non-deterministic depending on the object size and amount of clipping needed. While not a huge issue with wireframe rendering, when polygons are filled or textured the number of pixels that must be displayed goes up dramatically. When rasterization becomes a bottleneck, the only way to alleviate it, other than just increasing the rasterization engine's clock speed, is parallelism. However, increasing parallelism is not enough, because more memory bandwidth is required for each parallel rasterization element added. In this subsection, various methods that could be used to implement for parallel rasterization in the future are discussed.

In this thesis's design, a single physical frame buffer memory was used. The memory bandwidth is that of the single ZBT frame buffering memory. This will most certainly cause a bottleneck as resolutions increase and the number and size of graphic objects increase. One method to overcome this is to use memory interleaving. Memory interleaving divides the memory into multiple partitions (7). For example if the memory is divided into 16 partitions, then every forth pixel of every forth scan line can be stored in one partition, as shown in Figure 5.3.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | c | d | a | b | c | d |
| e | f | g | h | e | f | g | h |
| i | j | k | l | i | j | k | l |
| m | n | o | p | m | n | o | p |
| a | b | c | d | a | b | c | d |
| e | f | g | h | e | f | g | h |
| i | j | k | l | i | j | k | l |
| m | n | o | p | m | n | o | p |

Figure 5.3: Interleaved memory organization

The figure above illustrates a small cross section of a computer screen. Each letter represents a different memory bank. By dividing the screen this way, up to 16 pixels can be written or read in parallel. To implement this there must be 16 memory banks connected to an FPGA device or there must be one memory that supports bursting at 16 times the rate of the rasterization logic. Any combination of these would work as well, 4 physical memories with a clock 4 times the rate of the rasterization logic would also work. Using these methods requires alteration of any rasterization hardware to support parallel calculation of pixels for each memory bank.

Another method of memory parallelism is called contiguous partitioning. In contiguous partitioning, instead of each memory partition dealing with every other pixel or scan line, each memory partition is assigned an area of the screen. This is show in Figure 5.4 for a 2x2 contiguous partition.

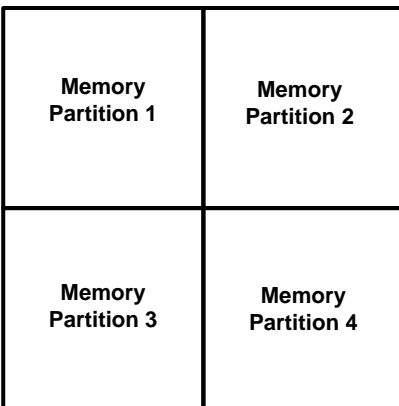


Figure 5.4: Contiguous Partitioning

In the figure above, $\frac{1}{4}$ of the screen is assigned to each memory partition. This implementation is simpler, but the performance increase is only seen if objects lie in different sections of the screen. If all the objects lie in one section of the screen no performance increase is observed.

In general, interleaved partitioning provides a better balance of the workload than contiguous partitioning. Use of either contiguous partitioning or interleaved partitioning involves creating multiple rasterization engines, one for each partition. This can dramatically increase the amount of hardware needed for rasterization but the performance increase can be as much 2x for each partition added.

5.2.6 Partial Reconfigurability

Xilinx FPGA development tools provide the ability to partially reconfigure a portion of the FPGA design during runtime. This type of operation is very similar to a processor using context switching. The only difference is that instead of loading different segments of software, it is actually hardware that is being loaded onto the design. In this sense, the FPGA can be used to time share resources among different functions.

In order to do this, Xilinx PlanAhead tool must be used. PlanAhead provides a means to floor plan a design and constrain certain parts of a design hierarchy into portions of an FPGA. For partially reconfigured designs special care must be taken to modularize the design. The design must be have a separate fixed portion and separate reconfigurable portions. Communication between modules is done via a bus macro called a TBUF. In addition, PlanAhead must be used to constrain each module into a vertical span or multiple vertical spans of a device. The bus macros must then be constrained to straddle the two columns that each module occupies. In addition, clocks that are needed for any permutation of the device must be present for all permutations.

This technique could be used to load different graphics processing hardware on the fly, as opposed to having hardware instantiated on the device at once. For instance, supposed a Xilinx FPGA is near its maximum logic utilization. If one application calls for a complex lighting model by no anti-aliasing, another needs a very simple lighting model but needs ant aliasing to prevent a jagged appearance. Assuming the design is made modular, the FPGA could in theory reconfigure itself for each application. For instance, application one could instantiate hardware for Phong shading with no anti-aliasing while application two could instantiate hardware for a simpler constant shading but with 4x anti-aliasing hardware acceleration. In a sense each application could partially reconfigure the FPGA for its own needs. This kind of functionality could be used to get the most functionality out of smaller scale FPGA devices.

APPENDIX A: GRAPHICS PIPELINE CONTROL REGISTERS

The graphics pipeline register interface is logic designed for communication between the Microblaze CPU and custom graphics pipeline logic over the PLB bus. This block contains various registers for configuration and debug of the GPU. The table below shows the registers present in both the 2D and 3D graphics engine.

Table A.1: GPU Configuration Register Memory Map and Register Definition

| Register Name | Type | Offset |
|-----------------------------------|---------------|---------------|
| ID Register | Read Only | 0x00 |
| Scratch Register | Read or Write | 0x04 |
| Status Register | Clear on Read | 0x08 |
| Enable Register | Read or Write | 0x0C |
| Background Color Register | Read or Write | 0x10 |
| Background Depth Register | Read or Write | 0x14 |
| ZBT Mailbox Write Register | Write Only | 0x40 |
| ZBT Mailbox Read Register | Write Only | 0x44 |
| ZBT Mailbox Data 0 Register | Read or Write | 0x48 |
| ZBT Mailbox Data 1 Register | Read or Write | 0x4C |
| Clipping Z Maximum Register | Read or Write | 0x340 |
| Matrix Column 0 Data Register | Read or Write | 0x380 |
| Matrix Column 1 Data Register | Read or Write | 0x384 |
| Matrix Column 2 Data Register | Read or Write | 0x388 |
| Matrix Column 3 Data Register | Read or Write | 0x38C |
| Matrix Write Enable | Write Only | 0x390 |
| Vector X Coordinate Data Register | Read or Write | 0x3C0 |
| Vector Y Coordinate Data Register | Read or Write | 0x3C4 |
| Vector Z Coordinate Data Register | Read or Write | 0x3C8 |
| Vector W Coordinate Data Register | Read or Write | 0x3CC |
| Vector Color Data Register | Read or Write | 0x3D0 |
| Vector Write Enable | Read or Write | 0x3D4 |

*Note all register dealing with the Z coordinate are not present in the 2D engine, these 3D only register are highlighted.

ID Register – 0x00

The Id register used to identify that this is indeed the GPU and also identify the current version of the GPU.

| Bits | Name | Type | Description |
|-------|-------------|-----------|--|
| 31:16 | GPU ID | Read Only | Reads back 0xBEEF |
| 15:0 | GPU Version | Read Only | Read back current version of GPU Currently 0x0006 |

Scratch Register – 0x04

Scratchpad register which can be written to any value. Used for PLB sanity check.

| Bits | Name | Type | Description |
|------|---------------|---------------|-----------------|
| 31:0 | Scratch Value | Read or Write | Write anything. |

Status Register – 0x08

The status register gives various debug and status information, each bit is described below.

| Bits | Name | Type | Description |
|------|------------------|---------------|---|
| 31:4 | Reserved | NA | Reserved |
| 8 | Object Complete | Clear on Read | This bit is set when the World Translation engine is complete with the translation matrix in use. |
| 7 | Input FIFO Full | Read Only | Gives the current status of the input FIFO of the GPU. This value should be read before pushing vector data into the GPU. |
| 6 | Arithmetic Error | Clear on Read | Divide by zero, Arithmetic Underflow, Arithmetic Overflow, or Invalid Operation has occurred in one of the floating point arithmetic engines. |
| 5 | FIFO Overflow | Clear on Read | A FIFO overflow in one of the designs FIFO has occurred. |
| 4 | FIFO Underflow | Clear on Read | A FIFO underflow in one of the designs FIFO has occurred. |
| 3 | End of Frame | Clear on Read | This bit is set when an end of frame even occurs. |
| 2 | ZBT Write Done | Clear on Read | A ZBT memory write has completed. |
| 1 | ZBT Read Valid | Clear on Read | A ZBT memory read has completed. |
| 0 | ZBT DCM Lock | Clear on Read | The ZBT DCM clock deskewing done. |

Enable Register – 0x0C

The Enable register is used to enable various block within the graphics pipeline.

| Bits | Name | Type | Description |
|-------|------------|---------------|---------------------------------------|
| 31:29 | Reserved | NA | Reserved |
| 1 | GPU Enable | Read or Write | Enables the GPU logic for use. |
| 0 | DVI Enable | Read or Write | Enables output to the DVI controller. |

Background Color Register – 0x10

This register sets the default background color of the screen.

| Bits | Name | Type | Description |
|-------|-------------|---------------|--------------------------------------|
| 17:12 | Red Value | Read or Write | Red Component of background color. |
| 11:6 | Green Value | Read or Write | Green Component of background color. |
| 5:0 | Blue Value | Read or Write | Blue Component of background color. |

Background Color Register – 0x14

This register sets the default background depth. This value is represented as a floating point 18.

| Bits | Name | Type | Description |
|------|-------------|---------------|-----------------------------|
| 17:0 | Depth Value | Read or Write | Floating point depth value. |

ZBT Mailbox Write Register – 0x40

This register is used to trigger a single write to the ZBT memory. The value written into this register is the address to be written to memory. Before this register is written, the ZBT Mailbox Data 0 and Data 1 registers must be set to the values to be written into memory.

| Bits | Name | Type | Description |
|------|---------------|------------|---------------------------|
| 31:0 | Write Address | Write Only | Address to be written to. |

ZBT Mailbox Read Register – 0x44

This register is used to trigger a single read to the ZBT memory. The value written into this register is the address to be read from memory. After this register is written, the ZBT Mailbox Data 0 and Data 1 registers hold the data which was read from memory.

| Bits | Name | Type | Description |
|------|--------------|------------|--------------------------|
| 31:0 | Read Address | Write Only | Address to be read from. |

ZBT Mailbox Data 0 Register – 0x48

This register is used to write bits 31 to 0 of ZBT memory when data is written to memory and is used to read bits 31 to 0 of ZBT memory when data is read.

| Bits | Name | Type | Description |
|------|--------------------|---------------|--|
| 31:0 | Read or Write Data | Read or Write | Data to be read from or written to memory. |

ZBT Mailbox Data 1 Register – 0x4C

This register is used to write bits 35 to 32 of ZBT memory when data is written to memory and is used to read bits 35 to 32 of ZBT memory when data is read.

| Bits | Name | Type | Description |
|------|--------------------|---------------|--|
| 31:4 | Reserved | NA | Reserved |
| 3:0 | Read or Write Data | Read or Write | Data to be read from or written to memory. |

Clipping Z Maximum Register – 0x340

The clipping z maximum register defines where the front face of the viewing volume is defined. All z values greater than the z maximum register are clipped.

| Bits | Name | Type | Description |
|------|-----------|---------------|---|
| 31:0 | Z Maximum | Read or Write | Defines the front face of the viewing volume. |

Matrix Column 0 Data Register – 0x380

This register is used to write floating point values into the geometric translation engines. This register is for column zero of the matrix.

| Bits | Name | Type | Description |
|------|-----------------|---------------|--|
| 31:0 | Matrix Column 0 | Read or Write | Geometric engine's column zero floating point value. |

Matrix Column 1 Data Register – 0x384

This register is used to write floating point values into the geometric translation engines.
This register is for column one of the matrix.

| Bits | Name | Type | Description |
|------|-----------------|---------------|---|
| 31:0 | Matrix Column 1 | Read or Write | Geometric engine's column one floating point value. |

Matrix Column 2 Data Register – 0x388

This register is used to write floating point values into the geometric translation engines.
This register is for column two of the matrix.

| Bits | Name | Type | Description |
|------|-----------------|---------------|---|
| 31:0 | Matrix Column 2 | Read or Write | Geometric engine's column two floating point value. |

Matrix Column 3 Data Register – 0x38C

This register is used to write floating point values into the geometric translation engines.
This register is for column three of the matrix.

| Bits | Name | Type | Description |
|------|-----------------|---------------|---|
| 31:0 | Matrix Column 3 | Read or Write | Geometric engine's column three floating point value. |

Matrix Write Enable Register – 0x390

This register triggers a write to a row of the geometric translation engine. The values of matrix column register 0 to 3 are written into the selected matrix column. Each geometric engine has four rows and there are four geometric engine blocks.

| Bits | Name | Type | Description |
|------|--------------------------|------------|---|
| 1:0 | Matrix Row Write Address | Write Only | Geometric engine column's write address. |
| 3:2 | Geometric Engine Select | Write Only | b00: World Translation Engine b01: View Translation Engine b10: Projection Translation Engine b11: Screen Translation Engine |

Vector X Coordinate Data Register – 0x3C0

This register is used to write floating point x value into the floating point engine.

| Bits | Name | Type | Description |
|------|--------------|---------------|--------------------------------------|
| 31:0 | X Coordinate | Read or Write | X Coordinate's floating point value. |

Vector Y Coordinate Data Register – 0x3C4

This register is used to write floating point y value into the floating point engine.

| Bits | Name | Type | Description |
|------|--------------|---------------|--------------------------------------|
| 31:0 | Y Coordinate | Read or Write | Y Coordinate's floating point value. |

Vector Z Coordinate Data Register – 0x3C8

This register is used to write floating point z value into the floating point engine.

| Bits | Name | Type | Description |
|------|--------------|---------------|--------------------------------------|
| 31:0 | Z Coordinate | Read or Write | Z Coordinate's floating point value. |

Vector W Coordinate Data Register – 0x3CC

This register is used to write floating point w value into the floating point engine.

| Bits | Name | Type | Description |
|------|--------------|---------------|--------------------------------------|
| 31:0 | W Coordinate | Read or Write | W Coordinate's floating point value. |

Vector Color Select Register – 0x3D0

This register sets the color of the vector. It takes two vectors to make a line, only the first vector is used as the color select. The second vector's color field is ignored.

| Bits | Name | Type | Description |
|-------|-------------|---------------|--------------------------------------|
| 17:12 | Red Value | Read or Write | Red Component of background color. |
| 11:6 | Green Value | Read or Write | Green Component of background color. |
| 5:0 | Blue Value | Read or Write | Blue Component of background color. |

Vector Write Enable Register – 0x3D4

This register triggers a push of a 3D vector into the graphics processing engine. The values of in the (x,y,z,w) coordinate register along with the color register are pushed into the graphic processors input FIFO.

| Bits | Name | Type | Description |
|-------------|-------------|-------------|--|
| 0 | Last Vector | Write Only | This bit indicates if this vector is the last vector input for this particular object. |

APPENDIX B: N64 CONTROLLER REGISTERS

The N64 Controller (20), designed by Nintendo for use with the Nintendo 64 console, is used for manipulation of 2D and 3D objects in the FPGA based graphics system. It is mainly a tool used for debugging and was selected based on availability. Using it, objects on the screen can be selected, translated, scaled and rotated. The N64 Controller registers are shown below

Table B.1: N64 Controller Interface Memory Map and Register Definition

| Register Name | Type | Offset |
|----------------------|-------------|---------------|
| Trigger | Write Only | 0x00 |
| Status | Read Only | 0x04 |
| Control Word Return | Read Only | 0x08 |
| Button Status Return | Read Only | 0x0C |

Trigger Register – 0x00

This register triggers commands to be sent to the N64 controller. Either a control word request or a button status request can be sent. There is also a reset bit for the internal state machine.

| Bits | Name | Type | Description |
|------|---------------------|------------|---|
| 31:3 | Reserved | NA | NA |
| 2 | Button Status Req. | Write Only | This bit sends a single cycle strobe to the controller state machine. The strobe triggers a serial bit stream to be sent to the N64 controller indicating that the button status be returned. |
| 1 | Control Word Req. | Write Only | This bit sends a single cycle strobe to the controller state machine. The strobe triggers a serial bit stream to be sent to the N64 controller indicating that the control word status be returned. |
| 0 | State Machine Reset | Write Only | Resets internal logic's state machine. Use this if the Controller Busy bit gets stuck high or if the controller is removed. |

Status Register – 0x04

This register provides general status for the N64 controller. A present bit can be polled to determine if the controller is present while a busy bit is used to determine when valid data from the controller has returned.

| Bits | Name | Type | Description |
|------|--------------------|-----------|--|
| 31:2 | Reserved | NA | NA |
| 1 | Controller Present | Read Only | Bit is high if controller is present on the line, else it is low. |
| 0 | Controller Busy | Read Only | After sending a control word or button status trigger this bit will be high until the control word or button status is returned by the controller. |

Control Word – 0x08

This register stores the contents of the last returned control word.

| Bits | Name | Type | Description |
|-------|-------------------|-----------|--|
| 31:24 | Reserved | NA | NA |
| 23:0 | Controller Status | Read Only | This is a 24 bit vector used to describe the status of the controller. Different bits are used to describe what peripheral is connected to the memory card slot. |

Control Word – 0x0C

This register stores the contents of the last returned button.

| Bits | Name | Type | Description |
|-------|-----------------|-----------|---|
| 31 | Button A | Read Only | High if A was pressed. |
| 30 | Button B | Read Only | High if B was pressed. |
| 29 | Button Z | Read Only | High if Z was pressed. |
| 28 | Button Start | Read Only | High if Start was pressed. |
| 27 | Joypad UP | Read Only | High if Up is pressed on Joypad. |
| 26 | Joypad Down | Read Only | High if Down is pressed on Joypad. |
| 25 | Joypad Left | Read Only | High if Left is pressed on Joypad. |
| 24 | Joypad Right | Read Only | High if Right is pressed on Joypad. |
| 23:22 | Unused | NA | NA |
| 21 | Button L | Read Only | High if L was pressed. |
| 20 | Button R | Read Only | High if R was pressed. |
| 19 | Cpad UP | Read Only | High if Up is pressed on Cpad |
| 18 | Cpad Down | Read Only | High if Down is pressed on Cpad |
| 17 | Cpad Left | Read Only | High if Left is pressed on Cpad |
| 16 | Cpad Right | Read Only | High if Right is pressed on Cpad |
| 15:8 | Joystick X-Axis | Read Only | 0-255 Value depending on placement of the X axis on the analog stick. |
| 7:0 | Joystick Y-Axis | Read Only | 0-255 Value depending on placement of the Y axis on the analog stick. |

APPENDIX C: VHDL SOURCE CODE

This appendix presents the VHDL code for the GPU. The Microblaze and floating point cores are not shown because they are proprietary IP provided by Xilinx. The code is listed in the sections below.

C.1 TOP LEVEL VHDL FILE

This VHDL file ties together the Microblaze CPU and the graphics processing custom logic. The tri state buffers for the iic, n64 controller and the ZBT memory are also defined here. In addition the DDR flops to drive data to the DVI controller are also instantiated.

```
-----  
--  Filename   : gpu_cpu_top.vhd  
--  
--  Date       : September 20 2007  
--  
--  Author     : James Warner  
--  
--  Desc       : GPU Top Level  
--  
-----  
  
library ieee;  
  use ieee.std_logic_1164.all;  
  use ieee.std_logic_arith.all;  
  use ieee.std_logic_unsigned.all;  
  
library work;  
  use work.gpu_pkg.all;  
  
library UNISIM;  
  use UNISIM.Vcomponents.all;  
  
entity gpu_cpu_top is
```

```

port(
-- Reset
ext_rst_n      : in   std_logic;

-- Main external clock reference
sys_clk       : in   std_logic;

-- UART (for debugging, stdio, etc.)
uart1_rx      : in   std_logic;
uart1_tx      : out  std_logic;

-- I2C Bus I/O (to be used later...DO NOT FORGET
-- PULLUP declarations on these pins in the UCF)
i2c_sda       : inout std_logic;
i2c_clk       : inout std_logic;
i2c_dvi_sda   : inout std_logic;
i2c_dvi_clk   : inout std_logic;

-- SRAM interface pins.
zbt_clk       : out  std_logic;
zbt_clk_fb    : in   std_logic;
zbt_addr      : out  std_logic_vector(20 downto 0);
zbt_dq        : inout std_logic_vector(35 downto 0);
zbt_ben       : out  std_logic_vector(3 downto 0);
zbt_oen       : out  std_logic;
zbt_cen       : out  std_logic;
zbt_wen       : out  std_logic;

-- Flash interface pins.
flash_clk     : in   std_logic;
flash_addr    : out  std_logic_vector(6 downto 0);
flash_data    : inout std_logic_vector(15 downto 0);
flash_cen     : out  std_logic;
flash_oen     : out  std_logic;
flash_wen     : out  std_logic;
flash_irq     : in   std_logic;

-- DDR DRAM interface pins
ddr_addr      : out  std_logic_vector(12 downto 0);
ddr_bankaddr  : out  std_logic_vector( 1 downto 0);
ddr_cas_n     : out  std_logic;
ddr_cke       : out  std_logic_vector( 1 downto 0);
ddr_cs_n      : out  std_logic_vector( 1 downto 0);
ddr_ras_n     : out  std_logic;
ddr_we_n      : out  std_logic;
ddr_dm        : out  std_logic_vector( 7 downto 0);
ddr_dqs       : inout std_logic_vector( 7 downto 0);
ddr_dqs_n     : inout std_logic_vector( 7 downto 0);
ddr_dq        : inout std_logic_vector(63 downto 0);
ddr_clk       : out  std_logic_vector( 1 downto 0);
ddr_clk_n     : out  std_logic_vector( 1 downto 0);
ddr_odt       : out  std_logic_vector( 1 downto 0);

-- Signals that drive the video DAC
dvi_reset_b   : out  std_logic;
dvi_xclk_n    : out  std_logic;
dvi_xclk_p    : out  std_logic;
dvi_d         : out  std_logic_vector(11 downto 0);
dvi_de        : out  std_logic;
dvi_gpio1     : inout std_logic;
dvi_h         : out  std_logic;
dvi_v         : out  std_logic;

```

```

-- N64 Controller input
n64_data      : inout std_logic
);
end gpu_cpu_top;

architecture hdl of gpu_cpu_top is

component cpu is
port (
  fpga_0_DDR2_SDRAM_DDR2_ODT_pin : out std_logic_vector(1 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_Addr_pin : out std_logic_vector(12 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_BankAddr_pin : out std_logic_vector(1 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_CAS_n_pin : out std_logic;
  fpga_0_DDR2_SDRAM_DDR2_CE_pin : out std_logic_vector(1 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_CS_n_pin : out std_logic_vector(1 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_RAS_n_pin : out std_logic;
  fpga_0_DDR2_SDRAM_DDR2_WE_n_pin : out std_logic;
  fpga_0_DDR2_SDRAM_DDR2_Clk_pin : out std_logic_vector(1 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_Clk_n_pin : out std_logic_vector(1 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_DM_pin : out std_logic_vector(7 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_DQS : inout std_logic_vector(7 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_DQS_n : inout std_logic_vector(7 downto 0);
  fpga_0_DDR2_SDRAM_DDR2_DQ : inout std_logic_vector(63 downto 0);
  fpga_0_SysAce_CompactFlash_SysAce_CEN_pin : out std_logic;
  fpga_0_SysAce_CompactFlash_SysAce_CLK_pin : in std_logic;
  fpga_0_SysAce_CompactFlash_SysAce_MPA_pin : out std_logic_vector(6
downto 0);
  fpga_0_SysAce_CompactFlash_SysAce_MPIRQ_pin : in std_logic;
  fpga_0_SysAce_CompactFlash_SysAce_MPD_pin : inout std_logic_vector(15
downto 0);
  fpga_0_SysAce_CompactFlash_SysAce_WEN_pin : out std_logic;
  fpga_0_SysAce_CompactFlash_SysAce_OEN_pin : out std_logic;
  fpga_0_GPU_INTF_REG_zbt_mbox_dval_pin : in std_logic;
  fpga_0_GPU_INTF_REG_zbt_mbox_wdone_pin : in std_logic;
  fpga_0_GPU_INTF_REG_zbt_mbox_rdata_pin : in std_logic_vector(35 downto
0);
  fpga_0_GPU_INTF_REG_zbt_mbox_wdata_pin : out std_logic_vector(35 downto
0);
  fpga_0_GPU_INTF_REG_zbt_mbox_addr_pin : out std_logic_vector(0 to 19);
  fpga_0_GPU_INTF_REG_zbt_mbox_we_pin : out std_logic;
  fpga_0_GPU_INTF_REG_zbt_mbox_sel_pin : out std_logic;
  fpga_0_GPU_INTF_REG_zbt_dcm_lock_pin : in std_logic;
  fpga_0_GPU_INTF_REG_video_enable_pin : out std_logic;
  fpga_0_GPU_INTF_REG_gpu_enable_pin : out std_logic;
  fpga_0_GPU_INTF_REG_background_pin : out std_logic_vector(35 downto 0);
  fpga_0_GPU_INTF_REG_zmax_o : out std_logic_vector(31 downto 0);
  fpga_0_GPU_INTF_REG_color_in_o : out std_logic_vector(17 downto 0);
  fpga_0_GPU_INTF_REG_x_in_o : out std_logic_vector(31 downto 0);
  fpga_0_GPU_INTF_REG_y_in_o : out std_logic_vector(31 downto 0);
  fpga_0_GPU_INTF_REG_z_in_o : out std_logic_vector(31 downto 0);
  fpga_0_GPU_INTF_REG_w_in_o : out std_logic_vector(31 downto 0);
  fpga_0_GPU_INTF_REG_point_trig_o : out std_logic;
  fpga_0_GPU_INTF_REG_matrix_we : out std_logic;
  fpga_0_GPU_INTF_REG_matrix_sel : out std_logic_vector(1 downto 0);
  fpga_0_GPU_INTF_REG_matrix_waddr : out std_logic_vector(1 downto 0);
  fpga_0_GPU_INTF_REG_matrix_wdata : out std_logic_vector(127 downto 0);
  fpga_0_GPU_INTF_REG_eof_pin : in std_logic;
  fpga_0_IIC_DVI_Sda_oen_pin : out std_logic;
  fpga_0_IIC_DVI_Sda_o_pin : out std_logic;
  fpga_0_IIC_DVI_Sda_i_pin : in std_logic;
  fpga_0_IIC_DVI_Scl_oen_pin : out std_logic;
  fpga_0_IIC_DVI_Scl_o_pin : out std_logic;
  fpga_0_IIC_DVI_Scl_i_pin : in std_logic;

```

```

fpga_0_IIC_EEPROM_Sda_oen_pin : out std_logic;
fpga_0_IIC_EEPROM_Sda_o_pin : out std_logic;
fpga_0_IIC_EEPROM_Sda_i_pin : in std_logic;
fpga_0_IIC_EEPROM_Scl_oen_pin : out std_logic;
fpga_0_IIC_EEPROM_Scl_o_pin : out std_logic;
fpga_0_IIC_EEPROM_Scl_i_pin : in std_logic;
fpga_0_RS232_Uart_1_TX_pin : out std_logic;
fpga_0_RS232_Uart_1_RX_pin : in std_logic;
sys_clk_pin : in std_logic;
sys_clk_125_pin : out std_logic;
sys_rst_pin : in std_logic;
dvi_clk_pin : out std_logic;
zbt_fb_clk_pin : in std_logic;
zbt_clk_pin : out std_logic;
n64_input_pin : in std_logic;
n64_highz_pin : out std_logic
);
end component;

component video_control_top is
port(
    -- Reset/Clock
    reset      : in std_logic; -- Async Reset.
    sys_clk    : in std_logic; -- System clock.
    zbt_clk    : in std_logic; -- ZBT memory clock.
    vga_clk    : in std_logic; -- Vga clock.

    -- VGA enable signals.
    vga_enable : in std_logic;

    -- New frame trigger signals.
    gpu_enable : in std_logic;

    -- Background
    background : in std_logic_vector(35 downto 0);

    -- End of file
    eof        : out std_logic;

    -- Matrix Memory access.
    matrix_we  : in std_logic;
    matrix_sel : in std_logic_vector(1 downto 0);
    matrix_waddr : in std_logic_vector(1 downto 0);
    matrix_wdata : in std_logic_vector(127 downto 0);

    -- Pixel Pipe.
    x_in       : in std_logic_vector(31 downto 0);
    y_in       : in std_logic_vector(31 downto 0);
    z_in       : in std_logic_vector(31 downto 0);
    w_in       : in std_logic_vector(31 downto 0);
    color      : in std_logic_vector(17 downto 0);
    pix_valid  : in std_logic;
    pix_ready  : out std_logic;

    -- Clipping Maximum
    zmax       : in std_logic_vector(31 downto 0);

    -- CPU Interface port.
    cpu_sel    : in std_logic;
    cpu_we     : in std_logic;
    cpu_addr   : in std_logic_vector(19 downto 0);
    cpu_wdata  : in std_logic_vector(35 downto 0);
    cpu_wdone  : out std_logic;

```

```

cpu_dval    : out std_logic;
cpu_rdata   : out std_logic_vector(35 downto 0);

-- ZBT interface
zbt_cen     : out std_logic;
zbt_wen     : out std_logic;
zbt_oen     : out std_logic;
zbt_ts      : out std_logic;
zbt_wdata   : out std_logic_vector(35 downto 0);
zbt_addr    : out std_logic_vector(17 downto 0);
zbt_rdata   : in  std_logic_vector(35 downto 0);

-- Dvi interface signals.
dvi_hsync_n : out std_logic;
dvi_vsync_n : out std_logic;
dvi_data_en : out std_logic;
dvi_data1   : out std_logic_vector(11 downto 0);
dvi_data2   : out std_logic_vector(11 downto 0)
);
end component;

-- Tieoffs to supplies...
signal vdd      : std_logic_vector(31 downto 0);
signal gnd      : std_logic_vector(31 downto 0);

signal reset    : std_logic;
signal reset_n  : std_logic;

signal dvi_hsync_n : std_logic;
signal dvi_vsync_n : std_logic;
signal dvi_data_en : std_logic;
signal dvi_data1   : std_logic_vector(11 downto 0);
signal dvi_data2   : std_logic_vector(11 downto 0);

signal i2c_eeprom_scl_i : std_logic;
signal i2c_eeprom_scl_o : std_logic;
signal i2c_eeprom_scl_oen : std_logic;
signal i2c_eeprom_sda_i : std_logic;
signal i2c_eeprom_sda_o : std_logic;
signal i2c_eeprom_sda_oen : std_logic;
signal i2c_dvi_scl_i : std_logic;
signal i2c_dvi_scl_o : std_logic;
signal i2c_dvi_scl_oen : std_logic;
signal i2c_dvi_sda_i : std_logic;
signal i2c_dvi_sda_o : std_logic;
signal i2c_dvi_sda_oen : std_logic;

signal zbt_cen_i : std_logic;
signal zbt_wen_i : std_logic;
signal zbt_oen_i : std_logic;
signal zbt_ts_i : std_logic;
signal zbt_addr_i : std_logic_vector(17 downto 0);
signal zbt_wdata_i : std_logic_vector(35 downto 0);
signal zbt_rdata_i : std_logic_vector(35 downto 0);

signal video_enable : std_logic;
signal gpu_enable   : std_logic;
signal background   : std_logic_vector(35 downto 0);
signal eof           : std_logic;
signal zbt_dcm_lock : std_logic;
signal zbt_mbox_sel : std_logic;
signal zbt_mbox_we  : std_logic;
signal zbt_mbox_addr : std_logic_vector(19 downto 0);

```

```

signal zbt_mbox_wdata      : std_logic_vector(35 downto 0);
signal zbt_mbox_rdata      : std_logic_vector(35 downto 0);
signal zbt_mbox_wdone      : std_logic;
signal zbt_mbox_dval       : std_logic;

signal vga_clk             : std_logic;
signal sys_clk_125         : std_logic;

signal color               : std_logic_vector(17 downto 0);
signal pix_valid          : std_logic;
signal pix_ready          : std_logic;

signal matrix_we          : std_logic;
signal matrix_sel         : std_logic_vector(1 downto 0);
signal matrix_waddr       : std_logic_vector(1 downto 0);
signal matrix_wdata       : std_logic_vector(127 downto 0);

signal x_in               : std_logic_vector(31 downto 0);
signal y_in               : std_logic_vector(31 downto 0);
signal z_in               : std_logic_vector(31 downto 0);
signal w_in               : std_logic_vector(31 downto 0);
signal color_in           : std_logic_vector(17 downto 0);
signal zmax               : std_logic_vector(31 downto 0);

signal n64_hi_ghz         : std_logic;

```

```
begin
```

```
-- Create both high and low polarity reset signals...
```

```
reset_n <= ext_rst_n;
reset   <= not(ext_rst_n);
```

```
-- Single bit ones and zeroes...
```

```
vdd <= (others => '1');
gnd <= (others => '0');
```

```
-- CPU instantiation.
```

```
u_cpu0 : cpu
  port map (
    fpga_0_RS232_Uart_1_RX_pin => uart1_rx,
    fpga_0_RS232_Uart_1_TX_pin => uart1_tx,
    fpga_0_DDR2_SDRAM_DDR2_ODT_pin => ddr_odt,
    fpga_0_DDR2_SDRAM_DDR2_Addr_pin => ddr_addr,
    fpga_0_DDR2_SDRAM_DDR2_BankAddr_pin => ddr_bankaddr,
    fpga_0_DDR2_SDRAM_DDR2_CAS_n_pin => ddr_cas_n,
    fpga_0_DDR2_SDRAM_DDR2_CE_pin => ddr_cke,
    fpga_0_DDR2_SDRAM_DDR2_CS_n_pin => ddr_cs_n,
    fpga_0_DDR2_SDRAM_DDR2_RAS_n_pin => ddr_ras_n,
    fpga_0_DDR2_SDRAM_DDR2_WE_n_pin => ddr_we_n,
    fpga_0_DDR2_SDRAM_DDR2_DM_pin => ddr_dm,
    fpga_0_DDR2_SDRAM_DDR2_DQS => ddr_dqs,
    fpga_0_DDR2_SDRAM_DDR2_DQS_n => ddr_dqs_n,
    fpga_0_DDR2_SDRAM_DDR2_DQ => ddr_dq,
    fpga_0_DDR2_SDRAM_DDR2_Clk_pin => ddr_clk,
    fpga_0_DDR2_SDRAM_DDR2_Clk_n_pin => ddr_clk_n,
    fpga_0_SysAce_CompactFlash_SysAce_CLK_pin => flash_clk,
    fpga_0_SysAce_CompactFlash_SysAce_MPA_pin => flash_addr,
    fpga_0_SysAce_CompactFlash_SysAce_MPIRQ_pin => flash_irq,
    fpga_0_SysAce_CompactFlash_SysAce_MPD_pin => flash_data,
    fpga_0_SysAce_CompactFlash_SysAce_CEN_pin => flash_cen,
    fpga_0_SysAce_CompactFlash_SysAce_WEN_pin => flash_wen,
    fpga_0_SysAce_CompactFlash_SysAce_OEN_pin => flash_oen,
    sys_clk_pin => sys_clk,
    sys_clk_125_pin => sys_clk_125,
```

```

sys_rst_pin          => ext_rst_n,
dvi_clk_pin         => vga_clk,
zbt_fb_clk_pin     => zbt_clk_fb,
zbt_clk_pin        => zbt_clk,
fpga_0_IIC_EEPROM_Scl_i_pin => i2c_eeprom_scl_i,
fpga_0_IIC_EEPROM_Scl_o_pin => i2c_eeprom_scl_o,
fpga_0_IIC_EEPROM_Scl_oen_pin => i2c_eeprom_scl_oen,
fpga_0_IIC_EEPROM_Sda_i_pin => i2c_eeprom_sda_i,
fpga_0_IIC_EEPROM_Sda_o_pin => i2c_eeprom_sda_o,
fpga_0_IIC_EEPROM_Sda_oen_pin => i2c_eeprom_sda_oen,
fpga_0_IIC_DVI_Scl_i_pin   => i2c_dvi_scl_i,
fpga_0_IIC_DVI_Scl_o_pin   => i2c_dvi_scl_o,
fpga_0_IIC_DVI_Scl_oen_pin => i2c_dvi_scl_oen,
fpga_0_IIC_DVI_Sda_i_pin   => i2c_dvi_sda_i,
fpga_0_IIC_DVI_Sda_o_pin   => i2c_dvi_sda_o,
fpga_0_IIC_DVI_Sda_oen_pin => i2c_dvi_sda_oen,
fpga_0_GPU_INTF_REG_videoe_enable_pin => videoe_enable,
fpga_0_GPU_INTF_REG_gpu_enable_pin   => gpu_enable,
fpga_0_GPU_INTF_REG_background_pin   => background,
fpga_0_GPU_INTF_REG_eof_pin          => eof,
fpga_0_GPU_INTF_REG_x_in_o           => x_in,
fpga_0_GPU_INTF_REG_y_in_o           => y_in,
fpga_0_GPU_INTF_REG_z_in_o           => z_in,
fpga_0_GPU_INTF_REG_w_in_o           => w_in,
fpga_0_GPU_INTF_REG_color_in_o       => color_in,
fpga_0_GPU_INTF_REG_matrix_we        => matrix_we,
fpga_0_GPU_INTF_REG_matrix_sel       => matrix_sel,
fpga_0_GPU_INTF_REG_matrix_waddr     => matrix_waddr,
fpga_0_GPU_INTF_REG_matrix_wdata    => matrix_wdata,
fpga_0_GPU_INTF_REG_pix_valid        => pix_valid,
fpga_0_GPU_INTF_REG_zmax_o           => zmax,
fpga_0_GPU_INTF_REG_zbt_dcm_lock_pin => zbt_dcm_lock,
fpga_0_GPU_INTF_REG_zbt_mbox_sel_pin => zbt_mbox_sel,
fpga_0_GPU_INTF_REG_zbt_mbox_we_pin  => zbt_mbox_we,
fpga_0_GPU_INTF_REG_zbt_mbox_addr_pin => zbt_mbox_addr,
fpga_0_GPU_INTF_REG_zbt_mbox_wdata_pin => zbt_mbox_wdata,
fpga_0_GPU_INTF_REG_zbt_mbox_rdata_pin => zbt_mbox_rdata,
fpga_0_GPU_INTF_REG_zbt_mbox_wdone_pin => zbt_mbox_wdone,
fpga_0_GPU_INTF_REG_zbt_mbox_dval_pin => zbt_mbox_dval,
n64_inpin           => n64_data,
n64_highz_pin      => n64_highz
);
u_gpu0 : video_control_top
port map (
  reset          => reset,
  sys_clk        => sys_clk_125,
  zbt_clk        => sys_clk_125,
  vga_clk        => vga_clk,
  vga_enable     => videoe_enable,
  gpu_enable     => gpu_enable,
  background     => background,
  matrix_we      => matrix_we,
  matrix_sel     => matrix_sel,
  matrix_waddr   => matrix_waddr,
  matrix_wdata   => matrix_wdata,
  x_in           => x_in,
  y_in           => y_in,
  z_in           => z_in,
  w_in           => w_in,
  color          => color_in,
  pix_valid      => pix_valid,
  pix_ready      => pix_ready,
  zmax           => zmax,
  eof            => eof,

```

```

    cpu_sel      => zbt_mbox_sel,
    cpu_we       => zbt_mbox_we,
    cpu_addr     => zbt_mbox_addr,
    cpu_wdata    => zbt_mbox_wdata,
    cpu_wdone    => zbt_mbox_wdone,
    cpu_dval     => zbt_mbox_dval,
    cpu_rdata    => zbt_mbox_rdata,
    zbt_cen      => zbt_cen_i,
    zbt_wen      => zbt_wen_i,
    zbt_oen      => zbt_oen_i,
    zbt_ts       => zbt_ts_i,
    zbt_wdata    => zbt_wdata_i,
    zbt_addr     => zbt_addr_i,
    zbt_rdata    => zbt_rdata_i,
    dvi_hsync_n => dvi_hsync_n,
    dvi_vsync_n => dvi_vsync_n,
    dvi_data_en  => dvi_data_en,
    dvi_data1    => dvi_data1,
    dvi_data2    => dvi_data2
);

-- Drive out the ZBT related signals.
zbt_addr      <= "000" & zbt_addr_i;
zbt_cen       <= zbt_cen_i;
zbt_wen       <= zbt_wen_i;
zbt_ben       <= (others => '0');
zbt_oen       <= zbt_oen_i;

zbt_dq        <= zbt_wdata_i when (zbt_ts_i = '0') else (others => 'Z');
zbt_rdata_i   <= zbt_dq;

-- Clock mirrors for the DAC clock
-- P clock...
xclk_p_mirror : ODDR -- Xilinx primitive...
port map (
    C => vga_clk,
    Q => dvi_xclk_p,
    CE => vdd(0),
    D1 => vdd(0),
    D2 => gnd(0),
    R => gnd(0),
    S => gnd(0)
);

-- N clock...
xclk_n_mirror : ODDR -- Xilinx primitive...
port map (
    C => vga_clk,
    Q => dvi_xclk_n,
    CE => vdd(0),
    D1 => gnd(0),
    D2 => vdd(0),
    R => gnd(0),
    S => gnd(0)
);

-- Vsync
vsync_mirror : ODDR -- Xilinx primitive...
port map (
    C => vga_clk,
    Q => dvi_v,
    CE => vdd(0),
    D1 => dvi_vsync_n,
    D2 => dvi_vsync_n,

```



```

    R => gnd(0),
    S => gnd(0)
);

-- Vsync
hsync_mirror : ODDR -- Xilinx primitive...
port map (
    C => vga_clk,
    Q => dvi_h,
    CE => vdd(0),
    D1 => dvi_hsync_n,
    D2 => dvi_hsync_n,
    R => gnd(0),
    S => gnd(0)
);

-- Vsync
de_mirror : ODDR -- Xilinx primitive...
port map (
    C => vga_clk,
    Q => dvi_de,
    CE => vdd(0),
    D1 => dvi_data_en,
    D2 => dvi_data_en,
    R => gnd(0),
    S => gnd(0)
);

-- Data bus (12 bit DDR)
dvi_ddr_data : for i in 11 downto 0 generate

    -- Data flop
    dvi_dbus_oddr : ODDR -- Xilinx primitive...
    port map (
        C => vga_clk,
        Q => dvi_d(i),
        CE => vdd(0),
        D1 => dvi_data1(i),
        D2 => dvi_data2(i),
        R => gnd(0),
        S => gnd(0)
    );

end generate;

-- Tieoffs for the VGA DAC. If these values are
-- not set as seen below, the rotten DAC will not
-- work.
dvi_gpio1    <= 'Z'; -- Keep this at Hi-Z just to be safe...
dvi_reset_b  <= '1'; -- Keep reset high as the DAC has a poweron reset.

-- IIC outputs.
i2c_sda <= 'Z' when (i2c_eeprom_sda_oen = '1') else i2c_eeprom_sda_o;
i2c_eeprom_sda_i <= i2c_sda;

i2c_clk <= 'Z' when (i2c_eeprom_scl_oen = '1') else i2c_eeprom_scl_o;
i2c_eeprom_scl_i <= i2c_clk;

i2c_dvi_sda <= 'Z' when (i2c_dvi_sda_oen = '1') else i2c_dvi_sda_o;
i2c_dvi_sda_i <= i2c_dvi_sda;

i2c_dvi_clk <= 'Z' when (i2c_dvi_scl_oen = '1') else i2c_dvi_scl_o;
i2c_dvi_scl_i <= i2c_dvi_clk;

```

```

n64_data <= 'Z' when (n64_highz = '1') else '0';
end hdl;

```

C.2 GRAPHICS PIPELINE TOP LEVEL VHDL FILE

This VHDL file ties together various components of the graphics pipeline. In particular, the frame buffer logic, VGA controller, ZBT memory interface, matrix multipliers, line rasterizer, and clipping logic.

```

-----
-- Filename   : video_control_top.vhd
--
-- Date       : November 2 2007
--
-- Author     : James Warner
--
-- Desc       : Top level of gpu controller.
-----

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_unsigned.all;

library work;
  use work.gpu_pkg.all;

entity video_control_top is
  port(
    -- Reset/Clock
    reset      : in std_logic; -- Async Reset.
    sys_clk    : in std_logic; -- System clock.
    zbt_clk    : in std_logic; -- ZBT memory clock.
    vga_clk    : in std_logic; -- Vga clock.

    -- VGA enable signals.
    vga_enable : in std_logic;

    -- New frame trigger signals.
    gpu_enable : in std_logic;

    -- Background
    background : in std_logic_vector(35 downto 0);

    -- End of file
    eof        : out std_logic;

    -- Matrix Memory access.

```

```

matrix_we    : in  std_logic;
matrix_sel   : in  std_logic_vector(1 downto 0);
matrix_waddr : in  std_logic_vector(1 downto 0);
matrix_wdata : in  std_logic_vector(127 downto 0);

-- Pixel Pipe.
x_in        : in  std_logic_vector(31 downto 0);
y_in        : in  std_logic_vector(31 downto 0);
z_in        : in  std_logic_vector(31 downto 0);
w_in        : in  std_logic_vector(31 downto 0);
color       : in  std_logic_vector(17 downto 0);
pix_valid   : in  std_logic;
pix_ready   : out std_logic;

-- Clipping Maximum
zmax        : in  std_logic_vector(31 downto 0);

-- CPU Interface port.
cpu_sel     : in  std_logic;
cpu_we      : in  std_logic;
cpu_addr    : in  std_logic_vector(19 downto 0);
cpu_wdata   : in  std_logic_vector(35 downto 0);
cpu_wdone   : out std_logic;
cpu_dval    : out std_logic;
cpu_rdata   : out std_logic_vector(35 downto 0);

-- ZBT interface
zbt_cen     : out std_logic;
zbt_wen     : out std_logic;
zbt_oen     : out std_logic;
zbt_ts      : out std_logic;
zbt_wdata   : out std_logic_vector(35 downto 0);
zbt_addr    : out std_logic_vector(17 downto 0);
zbt_rdata   : in  std_logic_vector(35 downto 0);

-- Dvi interface signals.
dvi_hsync_n : out std_logic;
dvi_vsync_n : out std_logic;
dvi_data_en : out std_logic;
dvi_data1   : out std_logic_vector(11 downto 0);
dvi_data2   : out std_logic_vector(11 downto 0)
);
end video_control_top;

architecture hdl of video_control_top is
  component float32_float18_conv is
    generic (
      NUM_OF_ENTRIES : integer := 1
    );
    port (
      -- Reset/Clock
      reset          : in  std_logic;
      clk            : in  std_logic;

      -- 32 single precision inputs
      float32_in_val : in  std_logic_vector(NUM_OF_ENTRIES-1 downto
0);
      float32_in_data : in  std_logic_vector(NUM_OF_ENTRIES*32-1 downto
0);

```

```

float18_out_val      : out std_logic_vector(NUM_OF_ENTRIES-1 downto
0);
float18_out_data     : out std_logic_vector(NUM_OF_ENTRIES*18-1 downto
0);
float18_out_underflow : out std_logic_vector(NUM_OF_ENTRIES-1 downto
0);
float18_out_overflow : out std_logic_vector(NUM_OF_ENTRIES-1 downto 0)
);
end component;

component vga_frame_reader is
port(
-- Clock, reset and enable signals
vga_clk      : in  std_logic;
reset_n      : in  std_logic;

-- Vga controller interface.
vga_fifo_afull : in  std_logic;
vga_data_val   : out std_logic;
vga_data       : out std_logic_vector(23 downto 0);
vga_eof        : in  std_logic;

-- ZBt memory interface.
mem_req        : out std_logic;
mem_afull      : in  std_logic;
mem_addr       : out std_logic_vector(18 downto 0);
mem_rpop       : out std_logic;
mem_rdata      : in  std_logic_vector(35 downto 0);
mem_reempty    : in  std_logic;
mem_rafull     : in  std_logic
);
end component;

component vga_ctrl is
generic(
-- Number of bits per color
NUM_COLOR_BITS : natural := 8;

-- Video timing Generics.
H_ACTIVE_VIDEO : integer := 640;
H_PULSE_LENGTH : integer := 96;
H_FRONT_PORCH  : integer := 16;
H_BACK_PORCH   : integer := 48;
V_ACTIVE_VIDEO : integer := 480;
V_PULSE_LENGTH : integer := 2;
V_FRONT_PORCH  : integer := 11;
V_BACK_PORCH   : integer := 31;

-- Pixel Fifo Generics
FIFO_DEPTH      : integer := 16;
FIFO_AFULL_THRESH : integer := 9;
FIFO_AEMPTY_THRESH: integer := 8
);
port(
-- Clock, reset and enable signals
vga_clk      : in  std_logic;
reset_n      : in  std_logic;
gpu_enable    : in  std_logic;

-- Input pixel data.

```

```

0);
    pixel_data_in    : in  std_logic_vector((NUM_COLOR_BITS * 3)-1 downto
0);
    pixel_wr_req     : in  std_logic;

    -- Signals to the display
    vsync_n          : out std_logic;
    hsync_n          : out std_logic;
    red_value        : out std_logic_vector(NUM_COLOR_BITS-1 downto 0);
    green_value      : out std_logic_vector(NUM_COLOR_BITS-1 downto 0);
    blue_value       : out std_logic_vector(NUM_COLOR_BITS-1 downto 0);
    vga_valid        : out std_logic;

    -- Display driver and fifo status
    pixel_fifo_full  : out std_logic;
    pixel_fifo_empty : out std_logic;
    pixel_fifo_afull : out std_logic;
    pixel_fifo_aempty : out std_logic;
    pixel_eof        : out std_logic;

);
end component;

component dvi_intf is
port (

    -- Clock, reset and enable signals
    clk          : in  std_logic;
    reset_n      : in  std_logic;
    enable       : in  std_logic;

    -- VGA input signals.
    vga_vsync_n  : in  std_logic;
    vga_hsync_n  : in  std_logic;
    vga_red      : in  std_logic_vector(7 downto 0);
    vga_green    : in  std_logic_vector(7 downto 0);
    vga_blue     : in  std_logic_vector(7 downto 0);
    vga_valid    : in  std_logic;

    -- Display driver and fifo status
    dvi_hsync_n  : out std_logic;
    dvi_vsync_n  : out std_logic;
    dvi_data_en  : out std_logic;
    dvi_dataa1   : out std_logic_vector(11 downto 0);
    dvi_dataa2   : out std_logic_vector(11 downto 0)

);
end component;

component zbt_frame_intf is

generic (
    ADDR_WIDTH      : integer := 20;
    BYTE_WIDTH      : integer := 9;
    DATA_WIDTH     : integer := 36
);
port (
    -- Reset/Clock
    reset           : in std_logic; -- Async Reset.
    sys_clk         : in std_logic; -- System clock.
    zbt_clk         : in std_logic; -- ZBT memory clock.
    vga_clk         : in std_logic; -- Vga clock.

```

```

-- New frame trigger signals.
gpu_enable : in std_logic;
vga_eof    : in std_logic;

-- VGA Read Only Port
vga_req    : in std_logic;
vga_afull  : out std_logic;
vga_addr   : in std_logic_vector(ADDR_WIDTH-2 downto 0);
vga_rpop   : in std_logic;
vga_rdata  : out std_logic_vector(DATA_WIDTH-1 downto 0);
vga_rempy  : out std_logic;
vga_rafull : out std_logic;

-- GPU Interface port
gpu_req    : in std_logic;
gpu_afull  : out std_logic;
gpu_size   : in std_logic_vector(1 downto 0);
gpu_addr   : in std_logic_vector(ADDR_WIDTH-2 downto 0);
gpu_rnw    : in std_logic;
gpu_wpush  : in std_logic;
gpu_wdata  : in std_logic_vector(DATA_WIDTH-1 downto 0);
gpu_wafull : out std_logic;
gpu_rpop   : in std_logic;
gpu_rdata  : out std_logic_vector(DATA_WIDTH-1 downto 0);
gpu_rdwaddr : out std_logic_vector(1 downto 0);
gpu_rempy  : out std_logic;

-- CPU Interface port.
cpu_sel    : in std_logic;
cpu_we     : in std_logic;
cpu_addr   : in std_logic_vector(ADDR_WIDTH-1 downto 0);
cpu_wdata  : in std_logic_vector(DATA_WIDTH-1 downto 0);
cpu_wdone  : out std_logic;
cpu_dval   : out std_logic;
cpu_rdata  : out std_logic_vector(DATA_WIDTH-1 downto 0);

-- ZBT interface
zbt_cen    : out std_logic;
zbt_wen    : out std_logic;
zbt_oen    : out std_logic;
zbt_ts     : out std_logic;
zbt_wdata  : out std_logic_vector(DATA_WIDTH-1 downto 0);
zbt_addr   : out std_logic_vector(ADDR_WIDTH-
bit_width(DATA_WIDTH/BYTE_WIDTH)-1 downto 0);
zbt_rdata  : in std_logic_vector(DATA_WIDTH-1 downto 0)
);
end component;

```

component matrix_transformation is

```

generic (
  NORMALIZE           : integer := 0;
  MATRIX_MULT_LATENCY : integer := 4;
  DIV_LATENCY         : integer := 16;
  FIFO_WIDTH          : integer := 32;
  FIFO_DEPTH          : integer := 16;
  FIFO_AFULL_THRESH  : integer := 8;
  FIFO_AEMPTY_THRESH : integer := 7;
  FIFO_FALL_THROUGH  : integer := 0
);
port (

```

```

-- Reset/Clock
reset : in std_logic;

```

```

clk      : in  std_logic;

-- Incomint points (18 bit floating point).
x_in     : in  std_logic_vector(17 downto 0);
y_in     : in  std_logic_vector(17 downto 0);
z_in     : in  std_logic_vector(17 downto 0);
w_in     : in  std_logic_vector(17 downto 0);
color_in : in  std_logic_vector(17 downto 0);
valid_in : in  std_logic;
pix_ready : out std_logic;

-- Matrix Access (18 bit floating point).
matrix_we  : in  std_logic;
matrix_waddr : in  std_logic_vector(1 downto 0);
matrix_wdata : in  std_logic_vector(71 downto 0);

-- Control
enable     : in  std_logic;
eof        : in  std_logic;

-- Output Integers, a line or edge.
x_out      : out std_logic_vector(17 downto 0);
y_out      : out std_logic_vector(17 downto 0);
z_out      : out std_logic_vector(17 downto 0);
w_out      : out std_logic_vector(17 downto 0);
color_out  : out std_logic_vector(17 downto 0);
valid_out  : out std_logic;
raster_rdy: in  std_logic
);
end component;

component line_creator is

generic (
  X_PIX_WIDTH : integer := 320;
  Y_PIX_WIDTH : integer := 240
);
port (

  -- Reset/Clock
  reset      : in  std_logic;
  clk        : in  std_logic;

  -- Incomint points (18 bit floating point).
  x_in       : in  std_logic_vector(17 downto 0);
  y_in       : in  std_logic_vector(17 downto 0);
  color_in   : in  std_logic_vector(17 downto 0);
  valid_in   : in  std_logic;
  pix_ready  : out std_logic;

  -- Control
  enable     : in  std_logic;
  eof        : in  std_logic;

  -- Output Integers, a line or edge.
  x0_out     : out std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
  y0_out     : out std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
  x1_out     : out std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
  y1_out     : out std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
  color_out  : out std_logic_vector(17 downto 0);
  valid_out  : out std_logic;
  raster_rdy: in  std_logic

);

```

```

end component;

component line_drawer is

  generic (
    X_PIX_WIDTH : integer := 320;
    Y_PIX_WIDTH : integer := 240
  );

  port (
    -- Reset/Clock
    reset      : in std_logic;
    clk        : in std_logic;

    -- Line Inputs.
    x0         : in std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto
0);
    x1         : in std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto
0);
    y0         : in std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto
0);
    y1         : in std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto
0);
    color      : in std_logic_vector(17 downto 0);
    pix_valid  : in std_logic;
    pix_ready  : out std_logic;

    -- Control
    background : in std_logic_vector(35 downto 0);
    enable     : in std_logic;
    eof        : in std_logic;

    -- Memory Outputs
    gpu_req    : out std_logic;
    gpu_rnw    : out std_logic;
    gpu_afull  : in std_logic;
    gpu_addr   : out std_logic_vector(18 downto 0);
    gpu_wpush  : out std_logic;
    gpu_wdata  : out std_logic_vector(35 downto 0);
    gpu_wafull : in std_logic
  );
end component;

```

```

component clipping_2d is

  port (
    -- Reset/Clock
    reset      : in std_logic;
    clk        : in std_logic;

    -- Control
    enable     : in std_logic;
    eof        : in std_logic;

    -- Incomint points (18 bit floating point).
    x_in       : in std_logic_vector(17 downto 0);
    y_in       : in std_logic_vector(17 downto 0);
    z_in       : in std_logic_vector(17 downto 0);
    color_in   : in std_logic_vector(17 downto 0);
    valid_in   : in std_logic;
    pix_in_ready : out std_logic;
  );

```



```

-- Window signals.
zmax      : in  std_logic_vector(17 downto 0);

-- Output Integers, a line or edge.
x_out     : out std_logic_vector(17 downto 0);
y_out     : out std_logic_vector(17 downto 0);
z_out     : out std_logic_vector(17 downto 0);
color_out : out std_logic_vector(17 downto 0);
valid_out : out std_logic;
pix_out_rdy : in  std_logic

);

end component;

signal reset_n          : std_logic;
signal vga_req          : std_logic;
signal vga_afull       : std_logic;
signal vga_addr        : std_logic_vector(18 downto 0);
signal vga_rpop        : std_logic;
signal vga_rdata       : std_logic_vector(35 downto 0);
signal vga_reempty     : std_logic;
signal vga_rafull      : std_logic;
signal vga_fifo_full   : std_logic;
signal vga_data_val    : std_logic;
signal vga_data        : std_logic_vector(23 downto 0);
signal vga_eof         : std_logic;
signal vga_vsync_n     : std_logic;
signal vga_hsync_n     : std_logic;
signal vga_red         : std_logic_vector(7 downto 0);
signal vga_green       : std_logic_vector(7 downto 0);
signal vga_blue        : std_logic_vector(7 downto 0);
signal vga_valid       : std_logic;
signal ground          : std_logic_vector(35 downto 0) := (others =>
'0');

signal vga_eof_meta    : std_logic;
signal vga_eof_sync    : std_logic;
signal vga_eof_sync_d  : std_logic;
signal sys_eof         : std_logic;

signal gpu_req         : std_logic;
signal gpu_rnw         : std_logic;
signal gpu_afull       : std_logic;
signal gpu_addr        : std_logic_vector(18 downto 0);
signal gpu_wpush       : std_logic;
signal gpu_wdata       : std_logic_vector(35 downto 0);
signal gpu_wafull     : std_logic;

signal float32_in_val_i : std_logic_vector(8 downto 0);
signal float32_in_data_i : std_logic_vector((9*32)-1 downto 0);
signal float18_out_val_i : std_logic_vector(8 downto 0);
signal float18_out_data_i : std_logic_vector((9*18)-1 downto 0);

signal zmax_float18    : std_logic_vector(17 downto 0);
signal matrix_wdata_float18 : std_logic_vector(71 downto 0);
signal y_in_float18    : std_logic_vector(17 downto 0);
signal x_in_float18    : std_logic_vector(17 downto 0);
signal z_in_float18    : std_logic_vector(17 downto 0);
signal w_in_float18    : std_logic_vector(17 downto 0);

signal matrix_weldly   : std_logic_vector(2 downto 0);
type matrix_sel_dly_t is array (0 to 2) of std_logic_vector(1 downto 0);
signal matrix_sel_dly  : matrix_sel_dly_t;

```

```

type matrix_waddr_dly_t is array (0 to 2) of std_logic_vector(1 downto 0);
signal matrix_waddr_dly      : matrix_waddr_dly_t;
signal matrix_world_we      : std_logic;
signal matrix_view_we       : std_logic;
signal matrix_projection_we  : std_logic;
signal matrix_screen_we     : std_logic;

signal pix_valid_dly        : std_logic_vector(2 downto 0);
type color_dly_t is array (0 to 2) of std_logic_vector(17 downto 0);
signal color_dly            : color_dly_t;

signal x_world              : std_logic_vector(17 downto 0);
signal y_world              : std_logic_vector(17 downto 0);
signal z_world              : std_logic_vector(17 downto 0);
signal w_world              : std_logic_vector(17 downto 0);
signal color_world         : std_logic_vector(17 downto 0);
signal valid_world         : std_logic;
signal pix_ready_world     : std_logic;

signal x_view               : std_logic_vector(17 downto 0);
signal y_view               : std_logic_vector(17 downto 0);
signal z_view               : std_logic_vector(17 downto 0);
signal w_view               : std_logic_vector(17 downto 0);
signal color_view          : std_logic_vector(17 downto 0);
signal valid_view          : std_logic;
signal pix_ready_view      : std_logic;

signal x_projection         : std_logic_vector(17 downto 0);
signal y_projection         : std_logic_vector(17 downto 0);
signal z_projection         : std_logic_vector(17 downto 0);
signal w_projection         : std_logic_vector(17 downto 0);
signal color_projection    : std_logic_vector(17 downto 0);
signal valid_projection    : std_logic;
signal pix_ready_projection : std_logic;

signal x_clipping          : std_logic_vector(17 downto 0);
signal y_clipping          : std_logic_vector(17 downto 0);
signal z_clipping          : std_logic_vector(17 downto 0);
signal w_clipping          : std_logic_vector(17 downto 0);
signal color_clipping      : std_logic_vector(17 downto 0);
signal valid_clipping      : std_logic;
signal pix_ready_clipping  : std_logic;

signal x_screen            : std_logic_vector(17 downto 0);
signal y_screen            : std_logic_vector(17 downto 0);
signal z_screen            : std_logic_vector(17 downto 0);
signal w_screen            : std_logic_vector(17 downto 0);
signal color_screen       : std_logic_vector(17 downto 0);
signal valid_screen       : std_logic;
signal pix_ready_screen   : std_logic;

signal x0_line_creator     : std_logic_vector(8 downto 0);
signal x1_line_creator     : std_logic_vector(8 downto 0);
signal y0_line_creator     : std_logic_vector(7 downto 0);
signal y1_line_creator     : std_logic_vector(7 downto 0);
signal color_line_creator  : std_logic_vector(17 downto 0);
signal valid_line_creator  : std_logic;
signal pix_ready_line_creator : std_logic;

signal pix_ready_line      : std_logic;

begin

reset_n <= not reset;

```

```

-- This process syncs and edge detects the eof signal
-- from the vga controller.
sys_eof_sync : process(sys_clk, reset)
begin
  if (reset = '1') then
    vga_eof_meta    <= '0';
    vga_eof_sync    <= '0';
    vga_eof_sync_d  <= '0';
    sys_eof         <= '0';

    elsif (sys_clk='1' and sys_clk' event) then

      vga_eof_meta    <= vga_eof;
      vga_eof_sync    <= vga_eof_meta;
      vga_eof_sync_d  <= vga_eof_sync;
      sys_eof <= '0';
      if (vga_eof_sync = '1' and vga_eof_sync_d = '0') then
        sys_eof <= '1';
      end if;

    end if;

  end process;
eof <= sys_eof;

-- This process decodes the selects from the matrix address
-- and sets the proper write enable for the matrix being edited.
-- I don't think this needs to be register, logic should be fast enough
-- at 100Mhz.
matrix_we_sel_dec : process(matrix_we_dly(2), matrix_sel_dly(2))
begin
  matrix_world_we    <= '0';
  matrix_view_we     <= '0';
  matrix_projecton_we <= '0';
  matrix_screen_we   <= '0';
  if (matrix_we_dly(2) = '1') then
    case matrix_sel_dly(2) is
      when "00" => matrix_world_we    <= '1';
      when "01" => matrix_view_we     <= '1';
      when "10" => matrix_projecton_we <= '1';
      when "11" => matrix_screen_we   <= '1';
      when others => NULL;
    end case;
  end if;
end process;

float32_float18_conv_0 : float32_float18_conv

  generic map (
    NUM_OF_ENTRIES => 9
  )
  port map (
    reset          => reset,
    clk            => sys_clk,
    float32_in_val => float32_in_val_i,
    float32_in_data => float32_in_data_i,
    float18_out_val => float18_out_val_i,
    float18_out_data => float18_out_data_i,
    float18_out_underflow => open,
    float18_out_overflow => open
  );

pix_val_id_dly_prc : process(sys_clk, reset)

```

```

begin
  if (reset = '1') then
    pix_valid_dly    <= (others => '0');
    color_dly        <= (others => (others => '0'));
    matrix_we_dly    <= (others => '0');
    matrix_waddr_dly <= (others => (others => '0'));
    matrix_sel_dly   <= (others => (others => '0'));
  else if (sys_clk = '1' and sys_clk'event) then
    pix_valid_dly(0) <= pix_valid;
    color_dly(0)     <= color;
    matrix_we_dly(0) <= matrix_we;
    matrix_waddr_dly(0) <= matrix_waddr;
    matrix_sel_dly(0) <= matrix_sel;
    for i in 1 to 2 loop
      pix_valid_dly(i) <= pix_valid_dly(i-1);
      color_dly(i)     <= color_dly(i-1);
      matrix_we_dly(i) <= matrix_we_dly(i-1);
      matrix_waddr_dly(i) <= matrix_waddr_dly(i-1);
      matrix_sel_dly(i) <= matrix_sel_dly(i-1);
    end loop;
  end if;
end process;

float32_in_val_i <= (others => '1');
float32_in_data_i <= zmax & matrix_wdata & w_in & z_in & y_in & x_in;

zmax_float18 <= float18_out_data_i((9*18)-1 downto 8*18);
matrix_wdata_float18 <= float18_out_data_i((8*18)-1 downto 4*18);
w_in_float18 <= float18_out_data_i((4*18)-1 downto 3*18);
z_in_float18 <= float18_out_data_i((3*18)-1 downto 2*18);
y_in_float18 <= float18_out_data_i((2*18)-1 downto 1*18);
x_in_float18 <= float18_out_data_i((1*18)-1 downto 0*18);

u_vga_ctrl_0 : vga_ctrl
  generic map (
    NUM_COLOR_BITS    => 8,
    H_ACTIVE_VIDEO    => 640,
    H_PULSE_LENGTH    => 96,
    H_FRONT_PORCH     => 16,
    H_BACK_PORCH      => 48,
    V_ACTIVE_VIDEO    => 480,
    V_PULSE_LENGTH    => 2,
    V_FRONT_PORCH     => 11,
    V_BACK_PORCH      => 31,
    FIFO_DEPTH        => 16,
    FIFO_AFULL_THRESH => 9,
    FIFO_AEMPTY_THRESH=> 8
  )
  port map(
    vga_clk          => vga_clk,
    reset_n          => reset_n,
    gpu_enable       => vga_enable,
    pixel_data_in    => vga_data,
    pixel_wr_req     => vga_data_val,
    vsync_n          => vga_vsync_n,
    hsync_n          => vga_hsync_n,
    red_value        => vga_red,
    green_value      => vga_green,
    blue_value       => vga_blue,
    vga_valid        => vga_valid,
    pixel_fifo_full  => open,
    pixel_fifo_empty => open,
    pixel_fifo_full  => vga_fifo_full,
    pixel_fifo_aempty=> open,

```

```

        pixel_eof      => vga_eof
    );

u_dvi_intf_0 : dvi_intf
    port map (
        clk            => vga_clk,
        reset_n       => reset_n,
        enable        => vga_enable,
        vga_vsync_n   => vga_vsync_n,
        vga_hsync_n   => vga_hsync_n,
        vga_red       => vga_red,
        vga_green     => vga_green,
        vga_blue      => vga_blue,
        vga_valid     => vga_valid,
        dvi_hsync_n   => dvi_hsync_n,
        dvi_vsync_n   => dvi_vsync_n,
        dvi_data_en   => dvi_data_en,
        dvi_data1     => dvi_data1,
        dvi_data2     => dvi_data2
    );

u_vga_frame_reader_0 : vga_frame_reader
    port map (
        vga_clk       => vga_clk,
        reset_n       => reset_n,
        vga_fifo_full => vga_fifo_full,
        vga_data_val  => vga_data_val,
        vga_data      => vga_data,
        vga_eof       => vga_eof,
        mem_req       => vga_req,
        mem_full      => vga_full,
        mem_addr      => vga_addr,
        mem_rpop      => vga_rpop,
        mem_rdata     => vga_rdata,
        mem_empty     => vga_empty,
        mem_rfull     => vga_rfull
    );

-- Send ready signal back to cpu.
pix_ready <= pix_ready_world;

world_translotion_0 : matrix_transformation

    generic map (
        NORMALIZE           => 0,
        MATRIX_MULT_LATENCY => 4,
        DIV_LATENCY        => 16,
        FIFO_WIDTH         => 18*5,
        FIFO_DEPTH         => 16,
        FIFO_AFULL_THRESH  => 8,
        FIFO_AEMPTY_THRESH => 7,
        FIFO_FALL_THROUGH  => 0
    )
    port map (
        reset            => reset,
        clk              => sys_clk,
        x_in             => x_in_float18,
        y_in             => y_in_float18,
        z_in             => z_in_float18,
        w_in             => w_in_float18,
        color_in        => color_dly(2),
        valid_in        => pix_valid_dly(2),
        pix_ready       => pix_ready_world,

```

```

matrix_we      => matrix_world_we,
matrix_waddr   => matrix_waddr_dly(2),
matrix_wdata   => matrix_wdata_float18,
enable        => gpu_enable,
eof           => sys_eof,
x_out         => x_world,
y_out         => y_world,
z_out         => z_world,
w_out         => w_world,
color_out     => color_world,
valid_out     => valid_world,
raster_rdy    => pix_ready_view
);

```

view_translation_0 : matrix_transformation

```

generic map (
  NORMALIZE           => 0,
  MATRIX_MULT_LATENCY => 4,
  DIV_LATENCY        => 16,
  FIFO_WIDTH         => 18*5,
  FIFO_DEPTH         => 16,
  FIFO_AFULL_THRESH  => 8,
  FIFO_AEMPTY_THRESH => 7,
  FIFO_FALL_THROUGH  => 0
)
port map (
  reset           => reset,
  clk            => sys_clk,
  x_in           => x_world,
  y_in           => y_world,
  z_in           => z_world,
  w_in           => w_world,
  color_in       => color_world,
  valid_in       => valid_world,
  pix_ready      => pix_ready_view,
  matrix_we      => matrix_view_we,
  matrix_waddr   => matrix_waddr_dly(2),
  matrix_wdata   => matrix_wdata_float18,
  enable        => gpu_enable,
  eof           => sys_eof,
  x_out         => x_view,
  y_out         => y_view,
  z_out         => z_view,
  w_out         => w_view,
  color_out     => color_view,
  valid_out     => valid_view,
  raster_rdy    => pix_ready_clipping
);

```

clipping_logic_0 : clipping_2d

```

port map (
  reset           => reset,
  clk            => sys_clk,
  enable        => gpu_enable,
  eof           => sys_eof,
  x_in           => x_view,
  y_in           => y_view,
  z_in           => z_view,
  color_in       => color_view,
  valid_in       => valid_view,
  pix_in_ready  => pix_ready_clipping,
  zmax          => zmax_float18,
  x_out         => x_clipping,

```

```

    y_out      => y_clipping,
    z_out      => z_clipping,
    color_out  => color_clipping,
    valid_out  => valid_clipping,
    pix_out_rdy => pix_ready_projection
);
w_clipping <= "00" & x"F000";

projection_translation_0 : matrix_transformation

```

```

generic map (
    NORMALIZE           => 0,
    MATRIX_MULT_LATENCY => 4,
    DIV_LATENCY        => 16,
    FIFO_WIDTH         => 18*5,
    FIFO_DEPTH         => 16,
    FIFO_AFULL_THRESH  => 8,
    FIFO_AEMPTY_THRESH => 7,
    FIFO_FALL_THROUGH  => 0
)
port map (
    reset      => reset,
    clk        => sys_clk,
    x_in       => x_clipping,
    y_in       => y_clipping,
    z_in       => z_clipping,
    w_in       => w_clipping,
    color_in   => color_clipping,
    valid_in   => valid_clipping,
    pix_ready  => pix_ready_projection,
    matrix_we  => matrix_projection_we,
    matrix_waddr => matrix_waddr_dly(2),
    matrix_wdata => matrix_wdata_float18,
    enable     => gpu_enable,
    eof        => sys_eof,
    x_out      => x_projection,
    y_out      => y_projection,
    z_out      => z_projection,
    w_out      => w_projection,
    color_out  => color_projection,
    valid_out  => valid_projection,
    raster_rdy => pix_ready_screen
);

```

```

screen_translation_0 : matrix_transformation

```

```

generic map (
    NORMALIZE           => 1,
    MATRIX_MULT_LATENCY => 4,
    DIV_LATENCY        => 16,
    FIFO_WIDTH         => 18*5,
    FIFO_DEPTH         => 16,
    FIFO_AFULL_THRESH  => 8,
    FIFO_AEMPTY_THRESH => 7,
    FIFO_FALL_THROUGH  => 0
)
port map (
    reset      => reset,
    clk        => sys_clk,
    x_in       => x_projection,
    y_in       => y_projection,
    z_in       => z_projection,
    w_in       => w_projection,
    color_in   => color_projection,

```

```

    valid_in    => valid_projection,
    pix_ready   => pix_ready_screen,
    matrix_we   => matrix_screen_we,
    matrix_waddr => matrix_waddr_dly(2),
    matrix_wdata => matrix_wdata_float18,
    enable      => gpu_enable,
    eof         => sys_eof,
    x_out       => x_screen,
    y_out       => y_screen,
    z_out       => z_screen,
    w_out       => w_screen,
    color_out   => color_screen,
    valid_out   => valid_screen,
    raster_rdy  => pix_ready_line_creator
);

```

line_creator_0 : line_creator

```

generic map (
    X_PIX_WIDTH => 320,
    Y_PIX_WIDTH => 240
)
port map (
    reset      => reset,
    clk        => sys_clk,
    x_in       => x_screen,
    y_in       => y_screen,
    color_in   => color_screen,
    valid_in   => valid_screen,
    pix_ready  => pix_ready_line_creator,
    enable     => gpu_enable,
    eof        => sys_eof,
    x0_out     => x0_line_creator,
    y0_out     => y0_line_creator,
    x1_out     => x1_line_creator,
    y1_out     => y1_line_creator,
    color_out  => color_line_creator,
    valid_out  => valid_line_creator,
    raster_rdy => pix_ready_line
);

```

line_drawler_0 : line_drawler

```

generic map (
    X_PIX_WIDTH => 320,
    Y_PIX_WIDTH => 240
)
port map (
    -- Reset/Clock
    reset      => reset,
    clk        => sys_clk,

    -- Line Inputs.
    x0         => x0_line_creator,
    x1         => x1_line_creator,
    y0         => y0_line_creator,
    y1         => y1_line_creator,
    color      => color_line_creator,
    pix_valid  => valid_line_creator,
    pix_ready  => pix_ready_line,

    -- Control

```



```

background    => background,
enable        => gpu_enable,
eof           => sys_eof,

-- Memory Outputs
gpu_req       => gpu_req,
gpu_rnw       => gpu_rnw,
gpu_afull     => gpu_afull,
gpu_addr      => gpu_addr,
gpu_wpush     => gpu_wpush,
gpu_wdata     => gpu_wdata,
gpu_wafull    => gpu_wafull

);

u_zbt_frame_intf_0 : zbt_frame_intf

generic map (
  ADDR_WIDTH => 20,
  BYTE_WIDTH => 9,
  DATA_WIDTH => 36
)
port map (
  reset        => reset,
  sys_clk      => sys_clk,
  zbt_clk      => zbt_clk,
  vga_clk      => vga_clk,

  -- New frame trigger signals.
  gpu_enable   => gpu_enable,
  vga_eof      => sys_eof,

  -- VGA Read Only Port
  vga_req      => vga_req,
  vga_afull    => vga_afull,
  vga_addr     => vga_addr,
  vga_rpop     => vga_rpop,
  vga_rdata    => vga_rdata,
  vga_reempty  => vga_reempty,
  vga_rafull   => vga_rafull,

  -- GPU Interface port
  gpu_req      => gpu_req,
  gpu_afull    => gpu_afull,
  gpu_size     => ground(1 downto 0),
  gpu_addr     => gpu_addr,
  gpu_rnw      => gpu_rnw,
  gpu_wpush    => gpu_wpush,
  gpu_wdata    => gpu_wdata,
  gpu_wafull   => gpu_wafull,
  gpu_rpop     => '0',
  gpu_rdata    => open,
  gpu_rdwaddr  => open,
  gpu_reempty  => open,

  -- CPU Interface port.
  cpu_sel      => cpu_sel,
  cpu_we       => cpu_we,
  cpu_addr     => cpu_addr,
  cpu_wdata    => cpu_wdata,
  cpu_wdone    => cpu_wdone,
  cpu_dval     => cpu_dval,
  cpu_rdata    => cpu_rdata,

```

```

-- ZBT interface
zbt_cen      => zbt_cen,
zbt_wen      => zbt_wen,
zbt_oen      => zbt_oen,
zbt_ts       => zbt_ts,
zbt_wdata    => zbt_wdata,
zbt_addr     => zbt_addr,
zbt_rdata    => zbt_rdata
);
end hdl;

```

C.3 MATRIX MULTIPLIER

This VHDL file implements a 4x4 18bit floating point matrix multiplication. It outputs a 18 bit 4x1 floating point vector every four clock cycles.

```

-----
--  Filename   :  matrix_mult4x4.vhd
--
--  Date       :  January 6th 2008
--
--  Author     :  James Warner
--
--  Desc       :  Matrix multiply of Nbit vector by NxN matrix.
--
--
--
--
--
--
--
--
-----

```

$$\begin{array}{c|c}
 \begin{array}{c} N1 \\ N2 \\ \cdot \\ NX \end{array} & * & \begin{array}{c|c}
 \begin{array}{cc} M11 & M21 & \cdot & \cdot & MX1 \\ M12 & M22 & \cdot & \cdot & MX2 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ M1Y & M2Y & \cdot & \cdot & \cdot \end{array}
 \end{array}
 \end{array}$$

```

-----

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library work;
use work.gpu_pkg.all;

entity matrix_mult4x4 is
port(
-- Clock, reset and enable signals
clk      : in  std_logic;
reset    : in  std_logic;

-- Vector Inputs.
vector_avail : in  std_logic;
vector_pop   : out std_logic;

```

```

vector_data      : in  std_logic_vector(18*4-1 downto 0);

-- Matrix Inputs.
matrix_we       : in  std_logic;
matrix_waddr    : in  std_logic_vector(bit_width(4)-1 downto 0);
matrix_wdata    : in  std_logic_vector(18*4-1 downto 0);
matrix_ready    : in  std_logic;

-- Matrix Outputs.
rslt_valid     : out std_logic;
rslt_data      : out std_logic_vector(18*4-1 downto 0);
rslt_bp        : in  std_logic
);
end matrix_mult4x4;

```

architecture hdl of matrix_mult4x4 is

```

constant MATRIX_SIZE : integer := 4;

```

```

-- Small distributed ram to store matrix.

```

```

component dist_ram_2port_1clk is

```

```

  generic (

```

```

    MEM_WIDTH : integer := 18;

```

```

    MEM_SIZE  : integer := MATRIX_SIZE*MATRIX_SIZE
  );

```

```

  port (

```

```

    -- Clock

```

```

    clk      : in  std_logic;

```

```

    -- Control signals

```

```

    we      : in  std_logic;

```

```

    waddr   : in  std_logic_vector(bit_width(MEM_SIZE)-1 downto 0);

```

```

    wdata   : in  std_logic_vector(MEM_WIDTH-1 downto 0);

```

```

    raddr   : in  std_logic_vector(bit_width(MEM_SIZE)-1 downto 0);

```

```

    rdata   : out std_logic_vector(MEM_WIDTH-1 downto 0)
  );

```

```

end component;

```

```

-- Floating point multiply.

```

```

component float18_mult IS

```

```

  port (

```

```

    a      : in  std_logic_vector(17 downto 0);

```

```

    b      : in  std_logic_vector(17 downto 0);

```

```

    operation_nd : in  std_logic;

```

```

    operation_rfd : out std_logic;

```

```

    clk      : in  std_logic;

```

```

    result   : out std_logic_vector(17 downto 0);

```

```

    underflow : out std_logic;

```

```

    overflow  : out std_logic;

```

```

    invalid_op : out std_logic;

```

```

    rdy      : out std_logic
  );

```

```

end component;

```

```

-- Floating point addition.

```

```

component float18_add IS

```

```

  port (

```

```

    a      : in  std_logic_vector(17 downto 0);

```

```

    b      : in  std_logic_vector(17 downto 0);

```

```

    operation_nd : in  std_logic;

```

```

    operation_rfd : out std_logic;

```

```

    clk      : in  std_logic;

```

```

    result   : out std_logic_vector(17 downto 0);
  );

```

```

        underflow      : out std_logic;
        overflow       : out std_logic;
        invalid_op     : out std_logic;
        rdy            : out std_logic
    );
end component;

type vector_state_type is (IDLE, MULT_VAL);
signal vector_state      : vector_state_type;
signal vector_pop_cnt    : std_logic_vector(bit_width(MATRIX_SIZE) - 1
downto 0);
signal fifo_rdy         : std_logic;
signal fifo_rdy_dly    : std_logic;

signal matrix_raddr     : std_logic_vector(bit_width(MATRIX_SIZE) - 1
downto 0);
signal matrix_rdata     : std_logic_vector(MATRIX_SIZE*18-1 downto 0);

signal mult0_en         : std_logic;
signal mult0_data1     : std_logic_vector(17 downto 0);
signal mult0_data2     : std_logic_vector(17 downto 0);
signal mult0_rslt      : std_logic_vector(17 downto 0);
signal mult0_underflow : std_logic;
signal mult0_overflow  : std_logic;
signal mult0_invalid_op : std_logic;
signal mult0_rdy       : std_logic;

signal mult1_en         : std_logic;
signal mult1_data1     : std_logic_vector(17 downto 0);
signal mult1_data2     : std_logic_vector(17 downto 0);
signal mult1_rslt      : std_logic_vector(17 downto 0);
signal mult1_underflow : std_logic;
signal mult1_overflow  : std_logic;
signal mult1_invalid_op : std_logic;
signal mult1_rdy       : std_logic;

signal mult2_en         : std_logic;
signal mult2_data1     : std_logic_vector(17 downto 0);
signal mult2_data2     : std_logic_vector(17 downto 0);
signal mult2_rslt      : std_logic_vector(17 downto 0);
signal mult2_underflow : std_logic;
signal mult2_overflow  : std_logic;
signal mult2_invalid_op : std_logic;
signal mult2_rdy       : std_logic;

signal mult3_en         : std_logic;
signal mult3_data1     : std_logic_vector(17 downto 0);
signal mult3_data2     : std_logic_vector(17 downto 0);
signal mult3_rslt      : std_logic_vector(17 downto 0);
signal mult3_underflow : std_logic;
signal mult3_overflow  : std_logic;
signal mult3_invalid_op : std_logic;
signal mult3_rdy       : std_logic;

signal add_0_1_en      : std_logic;
signal add_0_1_data1   : std_logic_vector(17 downto 0);
signal add_0_1_data2   : std_logic_vector(17 downto 0);
signal add_0_1_rslt    : std_logic_vector(17 downto 0);
signal add_0_1_underflow : std_logic;
signal add_0_1_overflow : std_logic;
signal add_0_1_invalid_op : std_logic;
signal add_0_1_rdy    : std_logic;

signal add_2_3_en      : std_logic;

```

```

signal add_2_3_data1      : std_logic_vector(17 downto 0);
signal add_2_3_data2      : std_logic_vector(17 downto 0);
signal add_2_3_rslt       : std_logic_vector(17 downto 0);
signal add_2_3_underflow  : std_logic;
signal add_2_3_overflow   : std_logic;
signal add_2_3_invalid_op : std_logic;
signal add_2_3_rdy       : std_logic;

signal add_0_1_2_3_en     : std_logic;
signal add_0_1_2_3_data1  : std_logic_vector(17 downto 0);
signal add_0_1_2_3_data2  : std_logic_vector(17 downto 0);
signal add_0_1_2_3_rslt   : std_logic_vector(17 downto 0);
signal add_0_1_2_3_underflow : std_logic;
signal add_0_1_2_3_overflow : std_logic;
signal add_0_1_2_3_invalid_op : std_logic;
signal add_0_1_2_3_rdy   : std_logic;

signal rslt_cnt           : std_logic_vector(bit_width(MATRIX_SIZE) - 1
downto 0);
signal rslt_data_i       : std_logic_vector(MATRIX_SIZE*18-1 downto 0);

begin

-- Small distributed ram to store matrix.
matrix_ram_0 : dist_ram_2port_1clk
generic map (
    MEM_WIDTH => 18*MATRIX_SIZE,
    MEM_SIZE  => MATRIX_SIZE
)
port map (
    clk      => clk,
    we       => matrix_we,
    waddr    => matrix_waddr,
    wdata    => matrix_wdata,
    raddr    => matrix_raddr,
    rdata    => matrix_rdata
);

-- Generate Multipliers.
mult_0 : float18_mult
port map (
    a          => mult0_data1,
    b          => mult0_data2,
    operation_nd => mult0_en,
    operation_rfd => open,
    clk        => clk,
    result     => mult0_rslt,
    underflow  => mult0_underflow,
    overflow   => mult0_overflow,
    invalid_op => mult0_invalid_op,
    rdy        => mult0_rdy
);

-- Generate Multipliers.
mult_1 : float18_mult
port map (
    a          => mult1_data1,
    b          => mult1_data2,
    operation_nd => mult1_en,
    operation_rfd => open,
    clk        => clk,
    result     => mult1_rslt,
    underflow  => mult1_underflow,
    overflow   => mult1_overflow,

```

```

        inval id_op    => mult1_inval id_op,
        rdy            => mult1_rdy
    );

-- Generate Multipliers.
mult_2 : float18_mult
port map (
    a            => mult2_data1,
    b            => mult2_data2,
    operation_nd => mult2_en,
    operation_rfd => open,
    clk          => clk,
    result       => mult2_rslt,
    underflow   => mult2_underflow,
    overflow     => mult2_overflow,
    inval id_op => mult2_inval id_op,
    rdy         => mult2_rdy
);

-- Generate Multipliers.
mult_3 : float18_mult
port map (
    a            => mult3_data1,
    b            => mult3_data2,
    operation_nd => mult3_en,
    operation_rfd => open,
    clk          => clk,
    result       => mult3_rslt,
    underflow   => mult3_underflow,
    overflow     => mult3_overflow,
    inval id_op => mult3_inval id_op,
    rdy         => mult3_rdy
);

-- Generate Adders.
adder_0_1 : float18_add
port map (
    a            => add_0_1_data1,
    b            => add_0_1_data2,
    operation_nd => add_0_1_en,
    operation_rfd => open,
    clk          => clk,
    result       => add_0_1_rslt,
    underflow   => add_0_1_underflow,
    overflow     => add_0_1_overflow,
    inval id_op => add_0_1_inval id_op,
    rdy         => add_0_1_rdy
);

-- Generate Adders.
adder_2_3 : float18_add
port map (
    a            => add_2_3_data1,
    b            => add_2_3_data2,
    operation_nd => add_2_3_en,
    operation_rfd => open,
    clk          => clk,
    result       => add_2_3_rslt,
    underflow   => add_2_3_underflow,
    overflow     => add_2_3_overflow,
    inval id_op => add_2_3_inval id_op,
    rdy         => add_2_3_rdy
);

```

```

-- Generate Adders.
adder_0_1_2_3 : float18_add
  port map (
    a          => add_0_1_2_3_data1,
    b          => add_0_1_2_3_data2,
    operation_nd => add_0_1_2_3_en,
    operation_rfd => open,
    clk        => clk,
    result     => add_0_1_2_3_rslt,
    underflow  => add_0_1_2_3_underflow,
    overflow   => add_0_1_2_3_overflow,
    invalid_op => add_0_1_2_3_invalid_op,
    rdy        => add_0_1_2_3_rdy
  );

ctrl_prc : process(clk, reset)
begin
  if (reset = '1') then

    vector_pop      <= '0';
    vector_pop_cnt  <= (others => '0');
    vector_state    <= IDLE;
    fifo_rdy        <= '0';
    fifo_rdy_dly    <= '0';

  elsif (clk = '1' and clk' event) then

    vector_pop      <= '0';
    fifo_rdy        <= '0';
    fifo_rdy_dly    <= fifo_rdy;

    case vector_state is
      when IDLE =>

        if (vector_avail = '1' and matrix_ready = '1' and rslt_bp = '0')
then
          vector_pop      <= '1';
          fifo_rdy        <= '1';
          vector_pop_cnt  <= (others => '0');
          vector_state    <= MULT_VAL;
        end if;

      when MULT_VAL =>

        fifo_rdy        <= '1';
        if (vector_pop_cnt = MATRIX_SIZE-1) then
          vector_pop_cnt <= (others => '0');
          if (vector_avail = '1' and matrix_ready = '1' and rslt_bp = '0')
then
            vector_pop      <= '1';
          else
            vector_state <= IDLE;
            fifo_rdy        <= '0';
          end if;
        else
          vector_pop_cnt <= vector_pop_cnt + 1;
        end if;

      end case;
    end if;
  end process;

```

```

matrix_mult_stage0_prc : process(clk, reset)
begin
    if (reset = '1') then
        mult0_en    <= '0';
        mult0_data1 <= (others => '0');
        mult0_data2 <= (others => '0');
        mult1_en    <= '0';
        mult1_data1 <= (others => '0');
        mult1_data2 <= (others => '0');
        mult2_en    <= '0';
        mult2_data1 <= (others => '0');
        mult2_data2 <= (others => '0');
        mult3_en    <= '0';
        mult3_data1 <= (others => '0');
        mult3_data2 <= (others => '0');
        matrix_raddr <= (others => '0');

    elsif (clk = '1' and clk'event) then

        mult0_en <= '0';
        mult1_en <= '0';
        mult2_en <= '0';
        mult3_en <= '0';

        if (fifo_rdy_dly = '1') then

            mult0_en    <= '1';
            mult0_data1 <= vector_data(17 downto 0);
            mult0_data2 <= matrix_rdata(17 downto 0);
            mult1_en    <= '1';
            mult1_data1 <= vector_data(35 downto 18);
            mult1_data2 <= matrix_rdata(35 downto 18);
            mult2_en    <= '1';
            mult2_data1 <= vector_data(53 downto 36);
            mult2_data2 <= matrix_rdata(53 downto 36);
            mult3_en    <= '1';
            mult3_data1 <= vector_data(71 downto 54);
            mult3_data2 <= matrix_rdata(71 downto 54);

            if (matrix_raddr = MATRIX_SIZE-1) then
                matrix_raddr <= (others => '0');
            else
                matrix_raddr <= matrix_raddr + 1;
            end if;

        end if;

    end if;
end process;

matrix_add_0_1_stage0_prc : process(clk, reset)
begin
    if (reset = '1') then
        add_0_1_en    <= '0';
        add_0_1_data1 <= (others => '0');
        add_0_1_data2 <= (others => '0');

    elsif (clk = '1' and clk'event) then
        add_0_1_en <= '0';

```



```

    if (mult0_rdy = '1' and mult1_rdy = '1') then
        add_0_1_en    <= '1';
        add_0_1_data1 <= mult0_rslt;
        add_0_1_data2 <= mult1_rslt;
    end if;

end if;

end process;

matrix_add_2_3_stage0_prc : process(clk, reset)
begin
    if (reset = '1') then
        add_2_3_en    <= '0';
        add_2_3_data1 <= (others => '0');
        add_2_3_data2 <= (others => '0');

    elsif (clk = '1' and clk'event) then
        add_2_3_en <= '0';

        if (mult2_rdy = '1' and mult3_rdy = '1') then
            add_2_3_en    <= '1';
            add_2_3_data1 <= mult2_rslt;
            add_2_3_data2 <= mult3_rslt;
        end if;

    end if;

end process;

matrix_add_0_1_2_3_stage0_prc : process(clk, reset)
begin
    if (reset = '1') then
        add_0_1_2_3_en    <= '0';
        add_0_1_2_3_data1 <= (others => '0');
        add_0_1_2_3_data2 <= (others => '0');

    elsif (clk = '1' and clk'event) then
        add_0_1_2_3_en <= '0';

        if (add_0_1_rdy = '1' and add_2_3_rdy = '1') then
            add_0_1_2_3_en    <= '1';
            add_0_1_2_3_data1 <= add_0_1_rslt;
            add_0_1_2_3_data2 <= add_2_3_rslt;
        end if;

    end if;

end process;

rslt_out_prc : process(clk, reset)
begin
    if (reset = '1') then
        rslt_valid    <= '0';
        rslt_data_i    <= (others => '0');

```

```

    rslt_cnt    <= (others => '0');
elseif (clk = '1' and clk'event) then
    rslt_valid <= '0';
    if (add_0_1_2_3_rdy = '1') then
        rslt_data_i(17 downto 0) <= add_0_1_2_3_rslt;
        for i in 1 to MATRIX_SIZE-1 loop
            rslt_data_i((18*(i+1))-1 downto 18*(i)) <= rslt_data_i((18*i)-1
downto 18*(i-1));
        end loop;

        if (rslt_cnt = MATRIX_SIZE-1) then
            rslt_cnt    <= (others => '0');
            rslt_valid <= '1';
        else
            rslt_cnt    <= rslt_cnt + 1;
        end if;

    end if;
end if;
end process;

rslt_data <= rslt_data_i;
end hdl;

```

C.4 MATRIX MULTIPLIER WITH BUFFERING AND NORMILIZATION

This VHDL file adds a input fifo for data buffering and dividers for division by w.

```

-----
--  Filename   :  matrix_transformation.vhd
--
--  Date       :  Febuary 27 2008
--
--  Author     :  James Warner
--
--  Desc       :  This block attaches a input fifo to a
--                4x4 matrix multiplication. An options
--                normilzation can be enable through
--                generi cs.
--
-----

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library work;

```

```

use work.gpu_pkg.all;

entity matrix_transformation is

generic (
    NORMALIZE          : integer := 0;
    MATRIX_MULT_LATENCY : integer := 4;
    DIV_LATENCY        : integer := 16;
    FIFO_WIDTH         : integer := 32;
    FIFO_DEPTH         : integer := 16;
    FIFO_AFULL_THRESH  : integer := 8;
    FIFO_AEMPTY_THRESH : integer := 7;
    FIFO_FALL_THROUGH  : integer := 0
);
port (
    -- Reset/Clock
    reset    : in  std_logic;
    clk      : in  std_logic;

    -- Incomint points (18 bit floating point).
    x_in     : in  std_logic_vector(17 downto 0);
    y_in     : in  std_logic_vector(17 downto 0);
    z_in     : in  std_logic_vector(17 downto 0);
    w_in     : in  std_logic_vector(17 downto 0);
    color_in : in  std_logic_vector(17 downto 0);
    valid_in : in  std_logic;
    pix_ready : out std_logic;

    -- Matrix Access (18 bit floating point).
    matrix_we    : in  std_logic;
    matrix_waddr : in  std_logic_vector(1 downto 0);
    matrix_wdata : in  std_logic_vector(71 downto 0);

    -- Control
    enable : in  std_logic;
    eof    : in  std_logic;

    -- Output Integers, a line or edge.
    x_out  : out std_logic_vector(17 downto 0);
    y_out  : out std_logic_vector(17 downto 0);
    z_out  : out std_logic_vector(17 downto 0);
    w_out  : out std_logic_vector(17 downto 0);
    color_out : out std_logic_vector(17 downto 0);
    valid_out : out std_logic;
    raster_rdy: in  std_logic
);
end matrix_transformation;

```

```

architecture hdl of matrix_transformation is

```

```

    component matrix_mult4x4
    port(
        -- Clock, reset and enable signals
        clk      : in  std_logic;
        reset    : in  std_logic;

        -- Vector Inputs.
        vector_avail : in  std_logic;
        vector_pop   : out std_logic;
    );
end component

```

```

vector_data      : in  std_logic_vector(18*4-1 downto 0);

-- Matrix Inputs.
matrix_we        : in  std_logic;
matrix_waddr     : in  std_logic_vector(bit_width(4)-1 downto 0);
matrix_wdata     : in  std_logic_vector(18*4-1 downto 0);
matrix_ready     : in  std_logic;

-- Matrix Outputs.
rslt_valid       : out std_logic;
rslt_data        : out std_logic_vector(18*4-1 downto 0);
rslt_bp          : in  std_logic
);

end component;

-- Floating point Division.
component float18_div is

port (
  a          : in  std_logic_vector(17 downto 0);
  b          : in  std_logic_vector(17 downto 0);
  operation_nd : in  std_logic;
  operation_rfd : out std_logic;
  clk        : in  std_logic;
  result     : out std_logic_vector(17 downto 0);
  underflow  : out std_logic;
  overflow   : out std_logic;
  invalid_op : out std_logic;
  divide_by_zero : out std_logic;
  rdy        : out std_logic
);

end component;

-- Single fifo clk.
component fifo_1clk is

generic (
  FIFO_WIDTH      : integer := 32;
  FIFO_DEPTH      : integer := 16;
  FIFO_AFULL_THRESH : integer := 8;
  FIFO_AEMPTY_THRESH: integer := 7;
  FIFO_FALL_THROUGH : integer := 0
);

port (
  -- Clock and reset
  reset : in  std_logic;
  clk   : in  std_logic;

  -- Control signals
  push  : in  std_logic;
  pop   : in  std_logic;

  -- Read write data
  wdata : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
  rdata : out std_logic_vector(FIFO_WIDTH-1 downto 0);

  -- Status flags.
  afull : out std_logic;
  aempty : out std_logic;
  empty : out std_logic;
  full  : out std_logic
);

```

```

);

end component;

signal vector_pop          : std_logic;
signal vector_data        : std_logic_vector(18*4-1 downto 0);
signal vector_avail       : std_logic;

signal pix_fifo_push      : std_logic;
signal pix_fifo_pop       : std_logic;
signal pix_fifo_wdata     : std_logic_vector(18*5-1 downto 0);
signal pix_fifo_rdata     : std_logic_vector(18*5-1 downto 0);
signal pix_fifo_afull     : std_logic;
signal pix_fifo_aempty    : std_logic;
signal pix_fifo_empty     : std_logic;
signal pix_fifo_full      : std_logic;

type color_pipe_dly_t is array (0 to MATRIX_MULT_LATENCY-1) of
std_logic_vector(17 downto 0);
signal color_pipe_dly    : color_pipe_dly_t;

signal divx_rslt          : std_logic_vector(17 downto 0);
signal divx_underflow    : std_logic;
signal divx_overflow     : std_logic;
signal divx_invalid_op   : std_logic;
signal divx_divide_by_zero : std_logic;
signal divx_rdy          : std_logic;
signal divy_rslt          : std_logic_vector(17 downto 0);
signal divy_underflow    : std_logic;
signal divy_overflow     : std_logic;
signal divy_invalid_op   : std_logic;
signal divy_divide_by_zero : std_logic;
signal divy_rdy          : std_logic;
signal divz_rslt          : std_logic_vector(17 downto 0);
signal divz_underflow    : std_logic;
signal divz_overflow     : std_logic;
signal divz_invalid_op   : std_logic;
signal divz_divide_by_zero : std_logic;
signal divz_rdy          : std_logic;
signal divw_rslt          : std_logic_vector(17 downto 0);
signal divvc_rslt         : std_logic_vector(17 downto 0);
type divvc_pipe_dly_t is array (0 to DIV_LATENCY-1) of std_logic_vector(17
downto 0);
signal divvc_pipe_dly    : divvc_pipe_dly_t;

signal rslt_valid        : std_logic;
signal rslt_data         : std_logic_vector(18*4-1 downto 0);
signal rslt_bp           : std_logic;

begin

-- Handle input fifo.
pix_ready    <= (not pix_fifo_afull);
pix_fifo_push <= valid_in;
pix_fifo_wdata <= color_in & x_in & y_in & z_in & w_in;
pix_fifo_pop  <= vector_pop;
vector_data   <= pix_fifo_rdata(18*4-1 downto 0);
vector_avail  <= not pix_fifo_empty;

-- Input fifo that buffers incoming vertices.
pix_fifo_0 : fifo_1clk

generic map(
    FIFO_WIDTH => FIFO_WIDTH,

```

```

    FIFO_DEPTH      => FIFO_DEPTH,
    FIFO_AFULL_THRESH => FIFO_AFULL_THRESH,
    FIFO_AEMPTY_THRESH => FIFO_AEMPTY_THRESH,
    FIFO_FALL_THROUGH => FIFO_FALL_THROUGH
)

port map (
  -- Clock and reset
  reset => reset,
  clk   => clk,

  -- Control signals
  push  => pix_fifo_push,
  pop   => pix_fifo_pop,

  -- Read write data
  wdata => pix_fifo_wdata,
  rdata => pix_fifo_rdata,

  -- Status flags.
  afull => pix_fifo_afull,
  aempty => pix_fifo_aempty,
  empty => pix_fifo_empty,
  full  => pix_fifo_full
);

-- Pipeline delay register for color.
color_pipe_dly_prc : process (clk, reset)
begin
  if (reset = '1') then
    color_pipe_dly <= (others => (others => '0'));
  elsif (clk = '1' and clk'event) then
    color_pipe_dly(0) <= pix_fifo_rdata(18*5-1 downto 18*4);
    for i in 1 to MATRIX_MULT_LATENCY-1 loop
      color_pipe_dly(i) <= color_pipe_dly(i-1);
    end loop;
  end if;
end process;

translation_matrix : matrix_mult4x4
port map (
  -- Clock, reset and enable signals
  clk       => clk,
  reset     => reset,

  -- Vector Inputs.
  vector_avail => vector_avail,
  vector_pop   => vector_pop,
  vector_data  => vector_data,

  -- Matrix Inputs.
  matrix_we    => matrix_we,
  matrix_waddr => matrix_waddr,
  matrix_wdata => matrix_wdata,
  matrix_ready => '1', -- I was born ready so I am always set.

  -- Matrix Outputs.
  rslt_valid  => rslt_valid,
  rslt_data   => rslt_data,
  rslt_bp     => rslt_bp
);
rslt_bp <= not raster_rdy;

```

```

disable_normalization : if (NORMALIZE = 0) generate
-- X result.
di_vx_rslt          <= rslt_data(18*4-1 downto 18*3);
di_vx_underflow    <= '0';
di_vx_overflow      <= '0';
di_vx_invalid_op    <= '0';
di_vx_divide_by_zero <= '0';
di_vx_rdy          <= rslt_valid;
-- Y result.
di_vy_rslt          <= rslt_data(18*3-1 downto 18*2);
di_vy_underflow    <= '0';
di_vy_overflow      <= '0';
di_vy_invalid_op    <= '0';
di_vy_divide_by_zero <= '0';
di_vy_rdy          <= rslt_valid;
-- Z result.
di_vz_rslt          <= rslt_data(18*2-1 downto 18*1);
di_vz_underflow    <= '0';
di_vz_overflow      <= '0';
di_vz_invalid_op    <= '0';
di_vz_divide_by_zero <= '0';
di_vz_rdy          <= rslt_valid;
-- W result.
di_vw_rslt          <= rslt_data(18*1-1 downto 18*0);
-- C result.
di_vc_rslt          <= color_pipe_delay(MATRIX_MULT_LATENCY-1);
di_vc_pipe_delay    <= (others => (others => '0'));
end generate;

```

```

enable_normalization : if (NORMALIZE = 1) generate

```

```

-- Floating point Division, x/w.
normalize_x : float18_div

```

```

port map (
a          => rslt_data(18*4-1 downto 18*3), -- x
b          => rslt_data(18*1-1 downto 18*0), -- w
operation_nd => rslt_valid,
operation_rfd => open,
clk        => clk,
result     => di_vx_rslt,
underflow  => di_vx_underflow,
overflow   => di_vx_overflow,
invalid_op => di_vx_invalid_op,
divide_by_zero => di_vx_divide_by_zero,
rdy       => di_vx_rdy
);

```

```

-- Floating point Division, y/w.
normalize_y : float18_div

```

```

port map (
a          => rslt_data(18*3-1 downto 18*2), -- y
b          => rslt_data(18*1-1 downto 18*0), -- w
operation_nd => rslt_valid,
operation_rfd => open,
clk        => clk,
result     => di_vy_rslt,
underflow  => di_vy_underflow,
overflow   => di_vy_overflow,
invalid_op => di_vy_invalid_op,
divide_by_zero => di_vy_divide_by_zero,
rdy       => di_vy_rdy
);

```

```

-- Floating point Division, z/w.
normalize_z : float18_div

port map (
  a          => rslt_data(18*2-1 downto 18*1), -- z
  b          => rslt_data(18*1-1 downto 18*0), -- w
  operation_nd => rslt_valid,
  operation_rfd => open,
  clk        => clk,
  result     => div_rslt,
  underflow  => div_underflow,
  overflow   => div_overflow,
  invalid_op => div_invalid_op,
  divide_by_zero => div_divide_by_zero,
  rdy        => div_rdy
);

-- w/w This should always be 1, you are dividing a number by itself.
div_w_rslt <= "00" & x"F000";

-- Delay the color to match up with the normalized result.
div_pipe_delay_prc : process (clk, reset)
begin
  if (reset = '1') then
    div_pipe_delay <= (others => (others => '0'));
  elsif (clk = '1' and clk'event) then
    div_pipe_delay(0) <= color_pipe_delay(MATRIX_MULT_LATENCY-1);
    for i in 1 to DIV_LATENCY-1 loop
      div_pipe_delay(i) <= div_pipe_delay(i-1);
    end loop;
  end if;
end process;
div_rslt <= div_pipe_delay(DIV_LATENCY-1);

end generate;

-- Wire up outputs.
x_out    <= div_rslt;
y_out    <= div_v_rslt;
z_out    <= div_z_rslt;
w_out    <= div_w_rslt;
color_out <= div_color_rslt;
valid_out <= div_rdy;

end hdl;

```

C.5 CLIPPING TREE

This VHDL file implements the planar intersection equations used in clipping.

```

-----
-- Filename   : clipping_tree.vhd
--
-- Date       : April 11 2008

```



```

--
-- Author      : James Warner
--
-- Desc       : A tree of floating point operations which
--             calculate a lines intersections with a rectangular boarder.
--             The equation is  $X = A + ((B - C) + (D - A)) / ((E - C) + /-(F - B))$ 
--
-----

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

```

```

library work;
use work.gpu_pkg.all;

```

```

entity clipping_tree is

```

```

    port (
        -- Reset/Clock
        reset      : in  std_logic;
        clk        : in  std_logic;

        -- Incomming operands.
        operand_a  : in  std_logic_vector(17 downto 0);
        operand_b  : in  std_logic_vector(17 downto 0);
        operand_c  : in  std_logic_vector(17 downto 0);
        operand_d  : in  std_logic_vector(17 downto 0);
        operand_e  : in  std_logic_vector(17 downto 0);
        operand_f  : in  std_logic_vector(17 downto 0);
        operand_dem_add : in  std_logic;
        operand_in_val  : in  std_logic;

        -- Outcoing operands
        operand_out   : out std_logic_vector(17 downto 0);
        operand_out_val : out std_logic
    );

```

```

end clipping_tree;

```

```

architecture hdl of clipping_tree is

```

```

    -- Floating point adder.
    component float18_add is
        port (
            a          : in  std_logic_vector(17 downto 0);
            b          : in  std_logic_vector(17 downto 0);
            operation_nd : in  std_logic;
            operation_rfd : out std_logic;
            clk        : in  std_logic;
            result     : out std_logic_vector(17 downto 0);
            underflow  : out std_logic;
            overflow   : out std_logic;
            invalid_op : out std_logic;
            rdy        : out std_logic
        );

```

```

    end component;

```

```

    -- Floating point multiplier.

```

```
component float18_mult is
```

```
port (  
  a          : in  std_logic_vector(17 downto 0);  
  b          : in  std_logic_vector(17 downto 0);  
  operation_nd : in  std_logic;  
  operation_rfd : out std_logic;  
  clk        : in  std_logic;  
  result     : out std_logic_vector(17 downto 0);  
  underflow  : out std_logic;  
  overflow   : out std_logic;  
  invalid_op : out std_logic;  
  rdy       : out std_logic  
);
```

```
end component;
```

```
-- Floating point divider.  
component float18_div is
```

```
port (  
  a          : in  std_logic_vector(17 downto 0);  
  b          : in  std_logic_vector(17 downto 0);  
  operation_nd : in  std_logic;  
  operation_rfd : out std_logic;  
  clk        : in  std_logic;  
  result     : out std_logic_vector(17 downto 0);  
  underflow  : out std_logic;  
  overflow   : out std_logic;  
  invalid_op : out std_logic;  
  divide_by_zero: out std_logic;  
  rdy       : out std_logic  
);
```

```
end component;
```

```
-- Floating point engine latency constants.
```

```
constant ADD_LATENCY      : integer := 8;  
constant MULT_LATENCY     : integer := 6;  
constant DIV_LATENCY      : integer := 17;  
constant A_LATENCY        : integer :=  
ADD_LATENCY+ADD_LATENCY+DIV_LATENCY;  
constant BC_MULT_DA_LATENCY : integer := ADD_LATENCY-MULT_LATENCY;
```

```
-- Clipping tree signals.
```

```
signal operand_b_add_c          : std_logic_vector(17 downto 0);  
signal operand_b_add_c_underflow : std_logic;  
signal operand_b_add_c_overflow  : std_logic;  
signal operand_b_add_c_invalid_op : std_logic;  
signal operand_b_add_c_rdy       : std_logic;  
signal operand_d_add_a          : std_logic_vector(17 downto 0);  
signal operand_d_add_a_underflow : std_logic;  
signal operand_d_add_a_overflow  : std_logic;  
signal operand_d_add_a_invalid_op : std_logic;  
signal operand_d_add_a_rdy       : std_logic;  
signal operand_e_add_c          : std_logic_vector(17 downto 0);  
signal operand_e_add_c_underflow : std_logic;  
signal operand_e_add_c_overflow  : std_logic;  
signal operand_e_add_c_invalid_op : std_logic;  
signal operand_e_add_c_rdy       : std_logic;  
signal operand_f_add_b          : std_logic_vector(17 downto 0);  
signal operand_f_add_b_underflow : std_logic;  
signal operand_f_add_b_overflow  : std_logic;  
signal operand_f_add_b_invalid_op : std_logic;
```

```

signal operand_f_add_b_rdy          : std_logic;
signal operand_bc_mult_da          : std_logic_vector(17 downto 0);
signal operand_bc_mult_da_underflow : std_logic;
signal operand_bc_mult_da_overflow : std_logic;
signal operand_bc_mult_da_invalid_op : std_logic;
signal operand_bc_mult_da_rdy      : std_logic;
signal operand_ec_add_fb          : std_logic_vector(17 downto 0);
signal operand_ec_add_fb_underflow : std_logic;
signal operand_ec_add_fb_overflow  : std_logic;
signal operand_ec_add_fb_invalid_op : std_logic;
signal operand_ec_add_fb_rdy      : std_logic;
signal operand_bcda_div_ecfb      : std_logic_vector(17 downto 0);
signal operand_bcda_div_ecfb_underflow : std_logic;
signal operand_bcda_div_ecfb_overflow : std_logic;
signal operand_bcda_div_ecfb_invalid_op : std_logic;
signal operand_bcda_div_ecfb_div_by_zero : std_logic;
signal operand_bcda_div_ecfb_rdy  : std_logic;
signal operand_out_underflow      : std_logic;
signal operand_out_overflow       : std_logic;
signal operand_out_invalid_op     : std_logic;
signal operand_neg_a              : std_logic_vector(17 downto 0);
signal operand_neg_b              : std_logic_vector(17 downto 0);
signal operand_b_or_negb          : std_logic_vector(17 downto 0);
signal operand_negc               : std_logic_vector(17 downto 0);
signal operand_f_add_b_add_sub    : std_logic_vector(17 downto 0);

-- Delay stages needed for pipelining calculation tree properly.
type operand_a_reg_t is array (0 to A_LATENCY-1) of std_logic_vector(17
downto 0);
signal operand_a_reg      : operand_a_reg_t;
type operand_bc_mult_da_t is array (0 to BC_MULT_DA_LATENCY-1) of
std_logic_vector(17 downto 0);
signal operand_bc_mult_da_reg : operand_bc_mult_da_t;
type operand_dem_add_t is array (0 to ADD_LATENCY-1) of std_logic;
signal operand_dem_add_reg : operand_dem_add_t;

begin

-- These need to be made negative for subtraction.
operand_neg_a <= (not operand_a(17)) & operand_a(16 downto 0);
operand_neg_b <= (not operand_b(17)) & operand_b(16 downto 0);
operand_negc <= (not operand_c(17)) & operand_c(16 downto 0);

-- Clipping floating point calculation tree.
-- Does the equivalent of:
-- A + (B-C) * (D-A)
-- -----
-- (E-C) +/- (F-B)
operand_b_or_negb <= operand_b when operand_dem_add = '0' else
operand_negb;
float_negb_or_b_add_negc : float18_add
port map (
a          => operand_b_or_negb,
b          => operand_negc,
operation_nd => operand_in_val,
operation_rfd => open,
clk        => clk,
result     => operand_b_add_c,
underflow  => operand_b_add_c_underflow,
overflow   => operand_b_add_c_overflow,
invalid_op => operand_b_add_c_invalid_op,
rdy        => operand_b_add_c_rdy
);

```

```

float_d_add_nega : float18_add
port map (
  a          => operand_d,
  b          => operand_nega,
  operation_nd => operand_in_val,
  operation_rfd => open,
  clk       => clk,
  result    => operand_d_add_a,
  underflow => operand_d_add_a_underflow,
  overflow  => operand_d_add_a_overflow,
  invalid_op => operand_d_add_a_invalid_op,
  rdy      => operand_d_add_a_rdy
);

float_e_add_negc : float18_add
port map (
  a          => operand_e,
  b          => operand_negc,
  operation_nd => operand_in_val,
  operation_rfd => open,
  clk       => clk,
  result    => operand_e_add_c,
  underflow => operand_e_add_c_underflow,
  overflow  => operand_e_add_c_overflow,
  invalid_op => operand_e_add_c_invalid_op,
  rdy      => operand_e_add_c_rdy
);

float_f_add_negb : float18_add
port map (
  a          => operand_f,
  b          => operand_negb,
  operation_nd => operand_in_val,
  operation_rfd => open,
  clk       => clk,
  result    => operand_f_add_b,
  underflow => operand_f_add_b_underflow,
  overflow  => operand_f_add_b_overflow,
  invalid_op => operand_f_add_b_invalid_op,
  rdy      => operand_f_add_b_rdy
);

float_bc_mult_da : float18_mult
port map (
  a          => operand_b_add_c,
  b          => operand_d_add_a,
  operation_nd => operand_f_add_b_rdy,
  operation_rfd => open,
  clk       => clk,
  result    => operand_bc_mult_da,
  underflow => operand_bc_mult_da_underflow,
  overflow  => operand_bc_mult_da_overflow,
  invalid_op => operand_bc_mult_da_invalid_op,
  rdy      => operand_bc_mult_da_rdy
);

delay_dem_add_prc : process(clk)
begin
  if (clk = '1' and clk'event) then
    operand_dem_add_reg(0) <= operand_dem_add;
    for i in 0 to ADD_LATENCY-2 loop
      operand_dem_add_reg(i+1) <= operand_dem_add_reg(i);
    end loop;
  end if;
end process;

```

```

end process;

operand_f_add_b_add_sub(16 downto 0) <= operand_f_add_b(16 downto 0);
operand_f_add_b_add_sub(17) <= operand_f_add_b(17) when
(operand_dem_add_reg(ADD_LATENCY-1) = '1') else
not operand_f_add_b(17);

float_ec_add_fb : float18_add
port map (
a => operand_e_add_c,
b => operand_f_add_b_add_sub,
operation_nd => operand_f_add_b_rdy,
operation_rfd => open,
clk => clk,
result => operand_ec_add_fb,
underflow => operand_ec_add_fb_underflow,
overflow => operand_ec_add_fb_overflow,
invalid_op => operand_ec_add_fb_invalid_op,
rdy => operand_ec_add_fb_rdy
);

delay_bc_mult_da_prc : process(clk)
begin
if (clk = '1' and clk'event) then
operand_bc_mult_da_reg(0) <= operand_bc_mult_da;
for i in 0 to BC_MULT_DA_LATENCY-2 loop
operand_bc_mult_da_reg(i+1) <= operand_bc_mult_da_reg(i);
end loop;
end if;
end process;

float_bcda_div_ecfb : float18_div

port map (
a => operand_bc_mult_da_reg(BC_MULT_DA_LATENCY-1),
b => operand_ec_add_fb,
operation_nd => operand_ec_add_fb_rdy,
operation_rfd => open,
clk => clk,
result => operand_bcda_div_ecfb,
underflow => operand_bcda_div_ecfb_underflow,
overflow => operand_bcda_div_ecfb_overflow,
invalid_op => operand_bcda_div_ecfb_invalid_op,
divide_by_zero=> operand_bcda_div_ecfb_div_by_zero,
rdy => operand_bcda_div_ecfb_rdy
);

delay_a_prc : process(clk)
begin
if (clk = '1' and clk'event) then
operand_a_reg(0) <= operand_a;
for i in 0 to A_LATENCY-2 loop
operand_a_reg(i+1) <= operand_a_reg(i);
end loop;
end if;
end process;

float_a_add_bcdaecfb : float18_add
port map (
a => operand_a_reg(A_LATENCY-1),
b => operand_bcda_div_ecfb,
operation_nd => operand_bcda_div_ecfb_rdy,
operation_rfd => open,
clk => clk,

```

```

    result      => operand_out,
    underflow  => operand_out_underflow,
    overflow   => operand_out_overflow,
    invalid_op => operand_out_invalid_op,
    rdy        => operand_out_val
);
end hdl;

```

C.6 OUTCODE GENERATOR

This VHDL file implements the outcodes used in Cohen-Sutherland clipping. These outcodes are used to determine a point's positions with respect to the clipping volume.

```

-----
-- Filename   : outcode_gen.vhd
--
-- Date       : April 11 2008
--
-- Author     : James Warner
--
-- Desc       : Creates a Cohen-Sutherland formatted
--              outcode used to determine the position
--              of a line's end point with respect to the
--              viewing window.
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library work;
use work.gpu_pkg.all;

entity outcode_gen is
port (
    -- Reset/Clock
    reset      : in  std_logic;
    clk        : in  std_logic;

    -- Incomint points (18 bit floating point).
    x_in       : in  std_logic_vector(17 downto 0);
    y_in       : in  std_logic_vector(17 downto 0);
    z_in       : in  std_logic_vector(17 downto 0);
    valid_in   : in  std_logic;

    -- Window signals.
    xmin       : in  std_logic_vector(17 downto 0);
    ymin       : in  std_logic_vector(17 downto 0);

```

```

    zmin      : in  std_logic_vector(17 downto 0);
    xmax      : in  std_logic_vector(17 downto 0);
    ymax      : in  std_logic_vector(17 downto 0);
    zmax      : in  std_logic_vector(17 downto 0);

    -- Output Integers, a line or edge.
    outcode   : out std_logic_vector(5 downto 0);
    valid_out : out std_logic

);

end outcode_gen;

architecture hdl of outcode_gen is

    -- Floating point comparitor.
    component float18_compare is

        port (
            a          : in  std_logic_vector(17 downto 0);
            b          : in  std_logic_vector(17 downto 0);
            operation  : in  std_logic_vector(5 downto 0);
            operation_nd : in  std_logic;
            operation_rfd : out std_logic;
            clk        : in  std_logic;
            result     : out std_logic_vector(0 downto 0);
            invalid_op : out std_logic;
            rdy        : out std_logic
        );

    end component;

    -- Compare constants for floating point comparitors.
    constant COMPARE_LESS_THAN      : std_logic_vector(5 downto 0) := "001100";
    constant COMPARE_GREATER_THAN   : std_logic_vector(5 downto 0) := "100100";
    constant COMPARE_LESS_EQUAL     : std_logic_vector(5 downto 0) := "011100";
    constant COMPARE_GREATER_EQUAL  : std_logic_vector(5 downto 0) := "110100";
    constant COMPARE_EQUAL          : std_logic_vector(5 downto 0) := "010100";
    constant COMPARE_NOT_EQUAL      : std_logic_vector(5 downto 0) := "101100";
    constant COMPARE_UNORDERED     : std_logic_vector(5 downto 0) := "000100";

    -- Outcode constants.
    constant OUTCODE_LEFT      : integer := 0;
    constant OUTCODE_RIGHT    : integer := 1;
    constant OUTCODE_BOTTOM   : integer := 2;
    constant OUTCODE_TOP      : integer := 3;
    constant OUTCODE_BEHIND   : integer := 4;
    constant OUTCODE_FRONT    : integer := 5;

    signal xmin_result      : std_logic_vector(0 downto 0);
    signal xmax_result     : std_logic_vector(0 downto 0);
    signal ymin_result     : std_logic_vector(0 downto 0);
    signal ymax_result     : std_logic_vector(0 downto 0);
    signal zmin_result     : std_logic_vector(0 downto 0);
    signal zmax_result     : std_logic_vector(0 downto 0);
    signal xmin_invalid_op : std_logic;
    signal xmax_invalid_op : std_logic;
    signal ymin_invalid_op : std_logic;
    signal ymax_invalid_op : std_logic;
    signal zmin_invalid_op : std_logic;
    signal zmax_invalid_op : std_logic;

begin

```

```
-- Check if coord is to the left of the viewport.  
xmin_float18_compare : float18_compare
```

```
port map (  
  a      => x_in,  
  b      => xmin,  
  operation      => COMPARE_LESS_THAN,  
  operation_nd   => valid_in,  
  operation_rfd  => open,  
  clk          => clk,  
  result        => xmin_result,  
  invalid_op    => xmin_invalid_op,  
  rdy          => valid_out  
);
```

```
-- Check if coord is to the right of the viewport.  
xmax_float18_compare : float18_compare
```

```
port map (  
  a      => x_in,  
  b      => xmax,  
  operation      => COMPARE_GREATER_THAN,  
  operation_nd   => valid_in,  
  operation_rfd  => open,  
  clk          => clk,  
  result        => xmax_result,  
  invalid_op    => xmax_invalid_op,  
  rdy          => open  
);
```

```
-- Check if coord is above of the viewport.  
ymin_float18_compare : float18_compare
```

```
port map (  
  a      => y_in,  
  b      => ymin,  
  operation      => COMPARE_LESS_THAN,  
  operation_nd   => valid_in,  
  operation_rfd  => open,  
  clk          => clk,  
  result        => ymin_result,  
  invalid_op    => ymin_invalid_op,  
  rdy          => open  
);
```

```
-- Check if coord is below of the viewport.  
ymax_float18_compare : float18_compare
```

```
port map (  
  a      => y_in,  
  b      => ymax,  
  operation      => COMPARE_GREATER_THAN,  
  operation_nd   => valid_in,  
  operation_rfd  => open,  
  clk          => clk,  
  result        => ymax_result,  
  invalid_op    => ymax_invalid_op,  
  rdy          => open  
);
```

```
-- Check if coord is above of the viewport.  
zmin_float18_compare : float18_compare
```



```

port map (
  a          => z_in,
  b          => zmin,
  operation  => COMPARE_LESS_THAN,
  operation_nd => valid_in,
  operation_rfd => open,
  clk       => clk,
  result    => zmin_result,
  invalid_op => zmin_invalid_op,
  rdy      => open
);

-- Check if coord is below of the viewport.
zmax_float18_compare : float18_compare

port map (
  a          => z_in,
  b          => zmax,
  operation  => COMPARE_GREATER_THAN,
  operation_nd => valid_in,
  operation_rfd => open,
  clk       => clk,
  result    => zmax_result,
  invalid_op => zmax_invalid_op,
  rdy      => open
);

outcode(OUTCODE_LEFT)    <= xmin_result(0);
outcode(OUTCODE_RIGHT)  <= xmax_result(0);
outcode(OUTCODE_BOTTOM) <= ymax_result(0);
outcode(OUTCODE_TOP)    <= ymin_result(0);
outcode(OUTCODE_FRONT)  <= zmax_result(0);
outcode(OUTCODE_BEHIND) <= zmin_result(0);

end hdl;

```

C.7 CLIPPING LOGIC

This VHDL file is the top level file for Cohen-Sutherland clipping. The output of this file is a clipped line.

```

-----
--  Filename   : clipping_2d.vhd
--
--  Date       : February 27 2008
--
--  Author     : James Warner
--
--  Desc       : Takes in 3-D projected floating point coordinates and
--               and clips them againts a given viewing volume.
--
-----

```

```

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_unsigned.all;

library work;
  use work.gpu_pkg.all;

entity clipping_2d is
  port (
    -- Reset/Clock
    reset      : in  std_logic;
    clk        : in  std_logic;

    -- Control
    enable     : in  std_logic;
    eof        : in  std_logic;

    -- Incomint points (18 bit floating point).
    x_in       : in  std_logic_vector(17 downto 0);
    y_in       : in  std_logic_vector(17 downto 0);
    z_in       : in  std_logic_vector(17 downto 0);
    color_in   : in  std_logic_vector(17 downto 0);
    valid_in   : in  std_logic;
    pix_in_ready : out std_logic;

    -- Window singals.
    zmax       : in  std_logic_vector(17 downto 0);

    -- Output Integers, a line or edge.
    x_out      : out std_logic_vector(17 downto 0);
    y_out      : out std_logic_vector(17 downto 0);
    z_out      : out std_logic_vector(17 downto 0);
    color_out  : out std_logic_vector(17 downto 0);
    valid_out  : out std_logic;
    pix_out_rdy : in  std_logic
  );
end clipping_2d;

architecture hdl of clipping_2d is
  component clipping_tree is
    port (
      -- Reset/Clock
      reset      : in  std_logic;
      clk        : in  std_logic;

      -- Incomming operands.
      operand_a  : in  std_logic_vector(17 downto 0);
      operand_b  : in  std_logic_vector(17 downto 0);
      operand_c  : in  std_logic_vector(17 downto 0);
      operand_d  : in  std_logic_vector(17 downto 0);
      operand_e  : in  std_logic_vector(17 downto 0);
      operand_f  : in  std_logic_vector(17 downto 0);
      operand_dem_add : in  std_logic;
      operand_in_val : in  std_logic;

      -- Outcoing operands
    );
  end component;

```

```

    operand_out      : out std_logic_vector(17 downto 0);
    operand_out_val  : out std_logic
);
end component;

component outcode_gen is
    port (
        -- Reset/Clock
        reset      : in  std_logic;
        clk        : in  std_logic;

        -- Incomint points (18 bit floating point).
        x_in       : in  std_logic_vector(17 downto 0);
        y_in       : in  std_logic_vector(17 downto 0);
        z_in       : in  std_logic_vector(17 downto 0);
        valid_in   : in  std_logic;

        -- Window signals.
        xmin       : in  std_logic_vector(17 downto 0);
        ymin       : in  std_logic_vector(17 downto 0);
        zmin       : in  std_logic_vector(17 downto 0);
        xmax       : in  std_logic_vector(17 downto 0);
        ymax       : in  std_logic_vector(17 downto 0);
        zmax       : in  std_logic_vector(17 downto 0);

        -- Output Integers, a line or edge.
        outcode    : out std_logic_vector(5 downto 0);
        valid_out  : out std_logic
    );
end component;

-- Fifo, for data buffering.
component fifo_1clk is
    generic (
        FIFO_WIDTH      : integer := 32;
        FIFO_DEPTH      : integer := 16;
        FIFO_AFULL_THRESH : integer := 8;
        FIFO_AEMPTY_THRESH: integer := 7;
        FIFO_FALL_THROUGH : integer := 0
    );
    port (
        -- Clock and reset
        reset : in  std_logic;
        clk   : in  std_logic;
        -- Control signals
        push  : in  std_logic;
        pop   : in  std_logic;
        -- Read write data
        wdata : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
        rdata : out std_logic_vector(FIFO_WIDTH-1 downto 0);
        -- Status flags.
        afull : out std_logic;
        aempty : out std_logic;
        empty : out std_logic;
        full  : out std_logic
    );
end component;

constant NEG_ONE      : std_logic_vector(17 downto 0) := "10" &
x"F000";

-- End of frame registers.

```

```

signal eof_dly          : std_logic;

signal zmax_reg        : std_logic_vector(17 downto 0);
signal zin_neg         : std_logic_vector(17 downto 0);

-- Input fifo signals.
signal fifo_input_push : std_logic;
signal fifo_input_pop  : std_logic;
signal fifo_input_wdata : std_logic_vector(18*4 + 6-1 downto 0);
signal fifo_input_rdata : std_logic_vector(18*4 + 6-1 downto 0);
signal fifo_input_afull : std_logic;
signal fifo_input_full  : std_logic;
signal fifo_input_aempty : std_logic;
signal fifo_input_empty : std_logic;

-- Accept fifo signals.
signal fifo_accept_push : std_logic;
signal fifo_accept_pop  : std_logic;
signal fifo_accept_wdata : std_logic_vector(18*4-1 downto 0);
signal fifo_accept_rdata : std_logic_vector(18*4-1 downto 0);
signal fifo_accept_afull : std_logic;
signal fifo_accept_full  : std_logic;
signal fifo_accept_aempty : std_logic;
signal fifo_accept_empty : std_logic;

-- Accept fifo signals.
signal fifo_clip_push : std_logic;
signal fifo_clip_pop  : std_logic;
signal fifo_clip_wdata : std_logic_vector(18*4 + 6-1 downto 0);
signal fifo_clip_rdata : std_logic_vector(18*4 + 6-1 downto 0);
signal fifo_clip_afull : std_logic;
signal fifo_clip_full  : std_logic;
signal fifo_clip_aempty : std_logic;
signal fifo_clip_empty : std_logic;

-- Accept store signals.
signal fifo_store_push : std_logic;
signal fifo_store_pop  : std_logic;
signal fifo_store_wdata : std_logic_vector(18*4 + 6-1 downto 0);
signal fifo_store_rdata : std_logic_vector(18*4 + 6-1 downto 0);
signal fifo_store_afull : std_logic;
signal fifo_store_full  : std_logic;
signal fifo_store_aempty : std_logic;
signal fifo_store_empty : std_logic;

-- Prefech signals.
signal fifo_store_x0 : std_logic_vector(17 downto 0);
signal fifo_store_y0 : std_logic_vector(17 downto 0);
signal fifo_store_z0 : std_logic_vector(17 downto 0);
signal fifo_store_color0 : std_logic_vector(17 downto 0);
signal fifo_store_outcode0 : std_logic_vector(5 downto 0);
signal fifo_store_x1 : std_logic_vector(17 downto 0);
signal fifo_store_y1 : std_logic_vector(17 downto 0);
signal fifo_store_z1 : std_logic_vector(17 downto 0);
signal fifo_store_color1 : std_logic_vector(17 downto 0);
signal fifo_store_outcode1 : std_logic_vector(5 downto 0);
signal fifo_store_rdy : std_logic;
signal fifo_store_done : std_logic;
type fifo_store_state_t is (POP_ZERO, POP_ONE, STORE, WAIT_FOR_DONE);
signal fifo_store_state : fifo_store_state_t;

-- Outcode signals.
signal outcode : std_logic_vector(5 downto 0);
signal outcode_valid : std_logic;

```

```

signal outcode_x      : std_logic_vector(17 downto 0);
signal outcode_y      : std_logic_vector(17 downto 0);
signal outcode_z      : std_logic_vector(17 downto 0);
signal outcode_color  : std_logic_vector(17 downto 0);

-- outcode latency signals.
constant OUTCODE_LATENCY : integer := 3;
type fifo_input_wdata_reg_t is array (0 to OUTCODE_LATENCY-1) of
std_logic_vector(18*4-1 downto 0);
signal fifo_input_wdata_reg : fifo_input_wdata_reg_t;

-- Decision state variables.
type decision_state_t is (IDLE, POP_FIRST, POP_LINE, PUSH_ACCEPT_LINE, REJECT_LINE, PUSH_CLIP_LINE);
signal decision_state : decision_state_t;
signal fifo_input_rdata0 : std_logic_vector(18*4 + 6 - 1 downto 0);

-- Clipping calculation tree signals.
signal operand_a0      : std_logic_vector(17 downto 0);
signal operand_b0      : std_logic_vector(17 downto 0);
signal operand_c0      : std_logic_vector(17 downto 0);
signal operand_d0      : std_logic_vector(17 downto 0);
signal operand_e0      : std_logic_vector(17 downto 0);
signal operand_f0      : std_logic_vector(17 downto 0);
signal operand_dem_add0 : std_logic;
signal operand_in_val0  : std_logic;
signal operand_out0     : std_logic_vector(17 downto 0);
signal operand_neg_out0 : std_logic_vector(17 downto 0);
signal operand_out_val0 : std_logic;

signal operand_a1      : std_logic_vector(17 downto 0);
signal operand_b1      : std_logic_vector(17 downto 0);
signal operand_c1      : std_logic_vector(17 downto 0);
signal operand_d1      : std_logic_vector(17 downto 0);
signal operand_e1      : std_logic_vector(17 downto 0);
signal operand_dem_add1 : std_logic;
signal operand_f1      : std_logic_vector(17 downto 0);
signal operand_in_val1  : std_logic;
signal operand_out1     : std_logic_vector(17 downto 0);
signal operand_neg_out1 : std_logic_vector(17 downto 0);
signal operand_out_val1 : std_logic;

-- Operand push signals.
type operand_push_state_t is (IDLE, POP_LINE, CALC);
signal operand_push_state : operand_push_state_t;
signal operand_push_cnt   : std_logic_vector(2 downto 0);
signal fifo_clip_rdata0   : std_logic_vector(18*4 + 6 - 1 downto 0);
signal fifo_clip_rdata1   : std_logic_vector(18*4 + 6 - 1 downto 0);

-- New outcode generation signals.
signal new_outcode_cnt   : std_logic_vector(2 downto 0);
signal new_x_in         : std_logic_vector(17 downto 0);
signal new_y_in         : std_logic_vector(17 downto 0);
signal new_z_in         : std_logic_vector(17 downto 0);
signal new_z_in_neg     : std_logic_vector(17 downto 0);
signal new_valid_in     : std_logic;
signal new_outcode      : std_logic_vector(5 downto 0);
signal new_outcode_valid : std_logic;
type new_outcode_xyz_reg_t is array (0 to OUTCODE_LATENCY-1) of
std_logic_vector(18*3-1 downto 0);
signal new_outcode_xyz_reg : new_outcode_xyz_reg_t;
signal new_outcode_x      : std_logic_vector(17 downto 0);
signal new_outcode_y      : std_logic_vector(17 downto 0);
signal new_outcode_z      : std_logic_vector(17 downto 0);

```

```

-- Pixel select state machine signals.
signal clip_pix_cnt      : std_logic_vector(2 downto 0);
signal clip_x0          : std_logic_vector(17 downto 0);
signal clip_y0          : std_logic_vector(17 downto 0);
signal clip_z0          : std_logic_vector(17 downto 0);
signal clip_color0     : std_logic_vector(17 downto 0);
signal clip_rdy0       : std_logic;
signal clip_x1         : std_logic_vector(17 downto 0);
signal clip_y1         : std_logic_vector(17 downto 0);
signal clip_z1         : std_logic_vector(17 downto 0);
signal clip_color1     : std_logic_vector(17 downto 0);
signal clip_rdy1       : std_logic;
signal clip_rdy        : std_logic;
signal clip_x0_reg     : std_logic_vector(17 downto 0);
signal clip_y0_reg     : std_logic_vector(17 downto 0);
signal clip_z0_reg     : std_logic_vector(17 downto 0);
signal clip_color0_reg : std_logic_vector(17 downto 0);
signal clip_x1_reg     : std_logic_vector(17 downto 0);
signal clip_y1_reg     : std_logic_vector(17 downto 0);
signal clip_z1_reg     : std_logic_vector(17 downto 0);
signal clip_color1_reg : std_logic_vector(17 downto 0);
signal clip_rdy_reg    : std_logic;
signal clip_done       : std_logic;

-- Arb state
type arb_state_t is (IDLE, CLIP0, CLIP1, ACCEPT0, ACCEPT1);
signal arb_state : arb_state_t;

```

```
begin
```

```

-- This process simply registers the min max values when and end of frame
-- arrives. This is so the process cannot change the value until the next
-- frame. Also, the min values signs are inverted to make subtraction
-- easier in the pipe stages below.
reg_register_prc : process(clk, reset)
begin
  if (reset = '1') then

    -- Reset all registers.
    zmax_reg <= (others => '0');
    eof_dly  <= '0';

  elsif (clk'event and clk = '1') then

    -- Line up eof with the latching of the registers.
    eof_dly <= eof;

    -- Latch registers on end of frame.
    if (eof = '1') then
      zmax_reg <= zmax;
    end if;

  end if;
end process;
z_in_neg(17)      <= not z_in(17);
z_in_neg(16 downto 0) <= z_in(16 downto 0);

-- Component generates Cohen-Sutherland outcode.
outcode_gen_0 : outcode_gen
  port map (
    reset    => reset,
    clk      => clk,
    x_in     => x_in,

```

```

    y_in      => y_in,
    z_in      => z_in,
    valid_in  => valid_in,
    xmin      => z_in,
    ymin      => z_in,
    zmin      => NEG_ONE,
    xmax      => z_in_neg,
    ymax      => z_in_neg,
    zmax      => zmax_reg,
    outcode   => outcode,
    valid_out => outcode_valid
);

-- Delay to match up x, y and color with outcode.
x_y_color_dly_prc : process (clk)
begin
    if (clk = '1' and clk'event) then
        fifo_input_wdata_reg(0) <= color_in & z_in & y_in & x_in;
        for i in 0 to OUTCODE_LATENCY-2 loop
            fifo_input_wdata_reg(i+1) <= fifo_input_wdata_reg(i);
        end loop;
    end if;
end process;
outcode_x      <= fifo_input_wdata_reg(OUTCODE_LATENCY-1)(17 downto 0);
outcode_y      <= fifo_input_wdata_reg(OUTCODE_LATENCY-1)(35 downto 18);
outcode_z      <= fifo_input_wdata_reg(OUTCODE_LATENCY-1)(53 downto 36);
outcode_color  <= fifo_input_wdata_reg(OUTCODE_LATENCY-1)(71 downto 54);

-- Input fifo assignments.
fifo_input_push <= outcode_valid;
fifo_input_wdata <= outcode & outcode_color & outcode_z & outcode_y &
outcode_x;
pix_in_ready    <= not fifo_input_afull;

-- Buffer incoming data.
input_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => 18*4 + 6,
    FIFO_DEPTH      => 16,
    FIFO_AFULL_THRESH => 8,
    FIFO_AEMPTY_THRESH => 1,
    FIFO_FALL_THROUGH => 1
)
port map (
    reset => reset,
    clk   => clk,
    push  => fifo_input_push,
    pop   => fifo_input_pop,
    wdata => fifo_input_wdata,
    rdata => fifo_input_rdata,
    afull => fifo_input_afull,
    aempty => fifo_input_aempty,
    empty => fifo_input_empty,
    full  => fifo_input_full
);

-- Control process for popping the input fifo and doing the outcode
-- comparison. The process looks at two points and determines weather.
-- To trivially accept, trivially reject, or do clipping. There are two
-- fifos, one for accepted lines and one for clipped lines. This way,
-- clipped lines cannot block accepted lines.
decision_ctrl_prc : process (clk, reset)
begin

```

```

if (reset = '1') then
    fifo_input_pop    <= '0';
    fifo_accept_push  <= '0';
    fifo_accept_wdata <= (others => '0');
    fifo_clip_push    <= '0';
    fifo_clip_wdata   <= (others => '0');
    decision_state    <= IDLE;
elsif (clk = '1' and clk'event) then

    -- Defaults.
    fifo_input_pop    <= '0';
    fifo_accept_push  <= '0';
    fifo_clip_push    <= '0';

    case decision_state is

        when IDLE =>

            -- Pop the first point and save it,
            -- We need two points to make a line!
            if (fifo_input_aempty = '0' and fifo_accept_afull = '0') then
                fifo_input_pop    <= '1';
                decision_state    <= POP_FIRST;
            end if;

            when POP_FIRST =>

                fifo_input_pop    <= '1';
                fifo_input_rdata0 <= fifo_input_rdata;
                decision_state    <= POP_LINE;

            when POP_LINE =>

                if ((fifo_input_rdata0(77 downto 72) or fifo_input_rdata(77 downto
72)) = x"0") then

                    -- If both points lie within the viewing window,
                    -- and the accept fifo is not almost full then
                    -- then push the stored pixel onto accept fifo and
                    -- store the new one, it will be pushed next cycle.
                    fifo_input_rdata0 <= fifo_input_rdata;
                    fifo_accept_push  <= '1';
                    fifo_accept_wdata <= fifo_input_rdata0(71 downto 0);
                    decision_state    <= PUSH_ACCEPT_LINE;

                elsif ((fifo_input_rdata0(77 downto 72) and fifo_input_rdata(77
downto 72)) /= x"0") then

                    -- If neither point intercepts the viewing window then
                    -- reject the line and do not push it on either fifo.
                    decision_state    <= REJECT_LINE;

                else

                    -- If the point can neither be trivially accepted or rejected then
                    -- send it to the clipping fifo when the clipping fifo is not
                    -- almost full. Push the stored pixel, the new one will be pushed
                    -- next cycle.
                    fifo_input_rdata0 <= fifo_input_rdata;
                    fifo_clip_push    <= '1';
                    fifo_clip_wdata   <= fifo_input_rdata0;
                    decision_state    <= PUSH_CLIP_LINE;

```



```

        end if;

when PUSH_ACCEPT_LINE =>

    -- Push second accepted point.
    fifo_accept_push  <= '1';
    fifo_accept_wdata <= fifo_input_rdata0(71 downto 0);

    -- Return to the idle state.
    decision_state   <= IDLE;

when REJECT_LINE      =>

    -- Return to the idle state.
    decision_state   <= IDLE;

when PUSH_CLIP_LINE   => null;

    -- Push second accepted point.
    fifo_clip_push   <= '1';
    fifo_clip_wdata  <= fifo_input_rdata0;

    -- Return to the idle state.
    decision_state   <= IDLE;

end case;

-- Just put this here incase as to not lock up the state machine.
if (eof = '1') then
    decision_state <= IDLE;
end if;

end if;
end process;

-- Buffer accept data.
accept_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => 18*4,
    FIFO_DEPTH      => 16,
    FIFO_AFULL_THRESH => 8,
    FIFO_AEMPTY_THRESH => 1,
    FIFO_FALL_THROUGH => 1
)
port map (
    reset => reset,
    clk   => clk,
    push  => fifo_accept_push,
    pop   => fifo_accept_pop,
    wdata => fifo_accept_wdata,
    rdata => fifo_accept_rdata,
    afull => fifo_accept_afull,
    aempty => fifo_accept_aempty,
    empty => fifo_accept_empty,
    full  => fifo_accept_full
);

-- Buffer clipping data.
clip_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => 18*4+6,
    FIFO_DEPTH      => 16,
    FIFO_AFULL_THRESH => 8,
    FIFO_AEMPTY_THRESH => 1,

```

```

    FIFO_FALL_THROUGH => 1
)
port map (
    reset => reset,
    clk   => clk,
    push  => fifo_clip_push,
    pop   => fifo_clip_pop,
    wdata => fifo_clip_wdata,
    rdata => fifo_clip_rdata,
    afull => fifo_clip_afull,
    aempty => fifo_clip_aempty,
    empty => fifo_clip_empty,
    full  => fifo_clip_full
);

-- Buffer clipping data.
-- This one needs to be big to handle clipping tree latency.
fifo_store_push <= fifo_clip_push;
fifo_store_wdata <= fifo_clip_wdata;
store_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => 18*4+6,
    FIFO_DEPTH      => 128,
    FIFO_AFULL_THRESH => 96,
    FIFO_AEMPTY_THRESH => 1,
    FIFO_FALL_THROUGH => 1
)
port map (
    reset => reset,
    clk   => clk,
    push  => fifo_store_push,
    pop   => fifo_store_pop,
    wdata => fifo_store_wdata,
    rdata => fifo_store_rdata,
    afull => fifo_store_afull,
    aempty => fifo_store_aempty,
    empty => fifo_store_empty,
    full  => fifo_store_full
);

-- Small state machine that prefetches data from storage fifo.
store_prefech_prc : process(reset, clk)
begin
    if (reset = '1') then
        fifo_store_pop      <= '0';
        fifo_store_x0      <= (others => '0');
        fifo_store_y0      <= (others => '0');
        fifo_store_z0      <= (others => '0');
        fifo_store_color0  <= (others => '0');
        fifo_store_outcode0 <= (others => '0');
        fifo_store_x1      <= (others => '0');
        fifo_store_y1      <= (others => '0');
        fifo_store_z1      <= (others => '0');
        fifo_store_color1  <= (others => '0');
        fifo_store_outcode1 <= (others => '0');
        fifo_store_rdy      <= '0';
        fifo_store_state    <= POP_ZERO;
    elsif (clk = '1' and clk'event) then

        -- Defaults
        fifo_store_pop <= '0';

        case fifo_store_state is

```

```

when POP_ZERO =>
    if (fifo_store_aempty = '0') then
        fifo_store_pop    <= '1';
        fifo_store_state  <= POP_ONE;
    end if;

when POP_ONE =>
    fifo_store_pop    <= '1';
    fifo_store_x0    <= fifo_store_rdata(17 downto 0);
    fifo_store_y0    <= fifo_store_rdata(35 downto 18);
    fifo_store_z0    <= fifo_store_rdata(53 downto 36);
    fifo_store_color0 <= fifo_store_rdata(71 downto 54);
    fifo_store_outcode0 <= fifo_store_rdata(77 downto 72);
    fifo_store_state  <= STORE;

    when STORE =>

        fifo_store_x1    <= fifo_store_rdata(17 downto 0);
        fifo_store_y1    <= fifo_store_rdata(35 downto 18);
        fifo_store_z1    <= fifo_store_rdata(53 downto 36);
        fifo_store_color1 <= fifo_store_rdata(71 downto 54);
        fifo_store_outcode1 <= fifo_store_rdata(77 downto 72);
        fifo_store_state  <= WAIT_FOR_DONE;
        fifo_store_rdy    <= '1';

    when WAIT_FOR_DONE =>

        if (fifo_store_done = '1') then
            fifo_store_rdy    <= '0';
            fifo_store_state  <= POP_ZERO;
        end if;

    end case;

end if;
end process;

-- This process pops, two points for the clipping fifo and push
-- the data through the clipping tree.
clip_fifo_pop_prc : process (reset, clk)
begin
    if (reset = '1') then
        fifo_clip_pop    <= '0';
        operand_a0       <= (others => '0');
        operand_b0       <= (others => '0');
        operand_c0       <= (others => '0');
        operand_d0       <= (others => '0');
        operand_e0       <= (others => '0');
        operand_f0       <= (others => '0');
        operand_dem_add0 <= '0';
        operand_in_val0  <= '0';
        operand_a1       <= (others => '0');
        operand_b1       <= (others => '0');
        operand_c1       <= (others => '0');
        operand_d1       <= (others => '0');
        operand_e1       <= (others => '0');
        operand_f1       <= (others => '0');
        operand_dem_add1 <= '0';
        operand_in_val1  <= '0';
        operand_push_state <= IDLE;
        operand_push_cnt <= (others => '0');
        fifo_clip_rdata0 <= (others => '0');
    end if;
end process;

```

```

    fifo_clip_rdata1 <= (others => '0');
elsif (clk = '1' and clk'event) then

    -- Defaults
    operand_in_val0 <= '0';
    operand_in_val1 <= '0';
    fifo_clip_pop    <= '0';

    case operand_push_state is

        when IDLE =>

            -- Pop off point x0, y0, z0
            if (fifo_clip_aempty = '0' and pix_out_rdy = '1') then
                fifo_clip_pop    <= '1';
                operand_push_state <= POP_LINE;
                operand_push_cnt  <= (others => '0');
            end if;

        when POP_LINE =>

            -- Pop off point x1, y1, z1
            fifo_clip_pop    <= '1';

            -- Save point x0, y0, z0
            fifo_clip_rdata0 <= fifo_clip_rdata;

            -- Cycle through all the sides.
            operand_push_state <= CALC;

        when CALC =>

            -- Increment counter.
            if (operand_push_cnt = "101") then
                operand_push_cnt <= (others => '0');
            else
                operand_push_cnt <= operand_push_cnt + 1;
            end if;

            -- Do bottom line intersection check.
            case operand_push_cnt is

                when "000" =>

                    -- Save point x1, y1, z1
                    fifo_clip_rdata1 <= fifo_clip_rdata;

                    -- Do x=z intersection check.
                    operand_in_val0 <= '1';
                    operand_a0      <= fifo_clip_rdata0(35 downto 18); -- y0
                    operand_b0      <= fifo_clip_rdata0(53 downto 36); -- z0
                    operand_c0      <= fifo_clip_rdata0(17 downto 0);  -- x0
                    operand_d0      <= fifo_clip_rdata(35 downto 18);  -- y1
                    operand_e0      <= fifo_clip_rdata(17 downto 0);   -- x1
                    operand_f0      <= fifo_clip_rdata(53 downto 36);  -- z1
                    operand_dem_add0 <= '0';

                    operand_in_val1 <= '1';
                    operand_a1      <= fifo_clip_rdata0(53 downto 36); -- z0
                    operand_b1      <= fifo_clip_rdata0(53 downto 36); -- z0
                    operand_c1      <= fifo_clip_rdata0(17 downto 0);  -- x0
                    operand_d1      <= fifo_clip_rdata(53 downto 36);  -- z1
            end case;
        end case;
    end if;
end process;

```

```

operand_e1      <= fifo_clip_rdata(17 downto 0);   -- x1
operand_f1      <= fifo_clip_rdata(53 downto 36); -- z1
operand_dem_add1 <= '0';

```

```
when "001" =>
```

```
-- Do x=-z intersection check.
```

```

operand_in_val0 <= '1';
operand_a0      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_b0      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_c0      <= fifo_clip_rdata0(17 downto 0);  -- x0
operand_d0      <= fifo_clip_rdata1(35 downto 18); -- y1
operand_e0      <= fifo_clip_rdata1(17 downto 0);  -- x1
operand_f0      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_dem_add0 <= '1';

```

```

operand_in_val1 <= '1';
operand_a1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_b1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_c1      <= fifo_clip_rdata0(17 downto 0);  -- x0
operand_d1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_e1      <= fifo_clip_rdata1(17 downto 0);  -- x1
operand_f1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_dem_add1 <= '1';

```

```
when "010" =>
```

```
-- Do the y=-z interscetion check.
```

```

operand_in_val0 <= '1';
operand_a0      <= fifo_clip_rdata0(17 downto 0);  -- x0
operand_b0      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_c0      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_d0      <= fifo_clip_rdata1(17 downto 0);  -- x1
operand_e0      <= fifo_clip_rdata1(35 downto 18); -- y1
operand_f0      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_dem_add0 <= '1';

```

```

operand_in_val1 <= '1';
operand_a1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_b1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_c1      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_d1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_e1      <= fifo_clip_rdata1(35 downto 18); -- y1
operand_f1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_dem_add1 <= '1';

```

```
when "011" =>
```

```
-- Do the y=z intersection check.
```

```

operand_in_val0 <= '1';
operand_a0      <= fifo_clip_rdata0(17 downto 0);  -- x0
operand_b0      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_c0      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_d0      <= fifo_clip_rdata1(17 downto 0);  -- x1
operand_e0      <= fifo_clip_rdata1(35 downto 18); -- y1
operand_f0      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_dem_add0 <= '0';

```

```

operand_in_val1 <= '1';
operand_a1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_b1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_c1      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_d1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_e1      <= fifo_clip_rdata1(35 downto 18); -- y1

```

```

operand_f1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_dem_add1 <= '0';

when "100" =>

-- Do z=-1 intersection check.
operand_in_val0 <= '1';
operand_a0      <= fifo_clip_rdata0(17 downto 0); -- x0
operand_b0      <= NEG_ONE; -- -1
operand_c0      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_d0      <= fifo_clip_rdata1(17 downto 0); -- x1
operand_e0      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_f0      <= NEG_ONE; -- -1
operand_dem_add0 <= '0';

operand_in_val1 <= '1';
operand_a1      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_b1      <= NEG_ONE; -- -1
operand_c1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_d1      <= fifo_clip_rdata1(35 downto 18); -- y1
operand_e1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_f1      <= NEG_ONE; -- -1
operand_dem_add1 <= '0';

when "101" =>

-- Do z=zmax intersection check.
operand_in_val0 <= '1';
operand_a0      <= fifo_clip_rdata0(17 downto 0); -- x0
operand_b0      <= zmax_reg; -- zmax
operand_c0      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_d0      <= fifo_clip_rdata1(17 downto 0); -- x1
operand_e0      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_f0      <= zmax_reg; -- zmax
operand_dem_add0 <= '0';

operand_in_val1 <= '1';
operand_a1      <= fifo_clip_rdata0(35 downto 18); -- y0
operand_b1      <= zmax_reg; -- zmax
operand_c1      <= fifo_clip_rdata0(53 downto 36); -- z0
operand_d1      <= fifo_clip_rdata1(35 downto 18); -- y1
operand_e1      <= fifo_clip_rdata1(53 downto 36); -- z1
operand_f1      <= zmax_reg; -- zmax
operand_dem_add1 <= '0';

-- Pop off point x0, y0, z0
if (fifo_clip_aempty = '0') then
  fifo_clip_pop <= '1';
  operand_push_state <= POP_LINE;
  operand_push_cnt <= (others => '0');
else
  operand_push_state <= IDLE;
end if;

when others => null; -- Silly VHDL, there are no more cases!

end case;

end case;

end if;
end process;

-- This clipping tree is calculates the intersection point of

```

```

-- points outside the viewing volume with the viewing volume itself.
clipping_tree_0 : clipping_tree
port map (
    reset          => reset,
    clk            => clk,
    operand_a      => operand_a0,
    operand_b      => operand_b0,
    operand_c      => operand_c0,
    operand_d      => operand_d0,
    operand_e      => operand_e0,
    operand_f      => operand_f0,
    operand_dem_add => operand_dem_add0,
    operand_in_val => operand_in_val0,
    operand_out     => operand_out0,
    operand_out_val => operand_out_val0
);
operand_neg_out0 <= (not operand_out0(17)) & operand_out0(16 downto 0);

clipping_tree_1 : clipping_tree
port map (
    reset          => reset,
    clk            => clk,
    operand_a      => operand_a1,
    operand_b      => operand_b1,
    operand_c      => operand_c1,
    operand_d      => operand_d1,
    operand_e      => operand_e1,
    operand_f      => operand_f1,
    operand_dem_add => operand_dem_add1,
    operand_in_val => operand_in_val1,
    operand_out     => operand_out1,
    operand_out_val => operand_out_val1
);
operand_neg_out1 <= (not operand_out1(17)) & operand_out1(16 downto 0);

-- Process the cycles through all 4 clipping calculation and pushes
-- the proper pixels into the new outcode calculation.
outcode_gen_prc : process (reset, clk)
begin
    if (reset = '1') then

        new_x_in      <= (others => '0');
        new_y_in      <= (others => '0');
        new_z_in      <= (others => '0');
        new_valid_in  <= '0';
        new_outcode_cnt <= (others => '0');

    elsif (clk = '1' and clk'event) then

        new_valid_in <= '0';
        if (operand_out_val0 = '1') then

            -- Set the outcode valid pulse.
            new_valid_in <= '1';

            -- Determine what portion of the volume intersection
            -- we are dealing with.
            case new_outcode_cnt is

                when "000" =>

                    -- This is the x=z intersection.
                    new_x_in <= operand_out1;
                    new_y_in <= operand_out0;

```

```

        new_z_in      <= operand_out1;
when "001" =>
    -- This is the x=-z intersection.
    new_x_in      <= operand_neg_out1;
    new_y_in      <= operand_out0;
    new_z_in      <= operand_out1;
when "010" =>
    -- This is the y=-z intersection.
    new_x_in      <= operand_out0;
    new_y_in      <= operand_neg_out1;
    new_z_in      <= operand_out1;
when "011" =>
    -- This is the y=z intersection
    new_x_in      <= operand_out0;
    new_y_in      <= operand_out1;
    new_z_in      <= operand_out1;
when "100" =>
    -- This is the z=-1 intersection
    new_x_in      <= operand_out0;
    new_y_in      <= operand_out1;
    new_z_in      <= NEG_ONE;
when "101" =>
    -- This is the z=zmax intersection
    new_x_in      <= operand_out0;
    new_y_in      <= operand_out1;
    new_z_in      <= zmax_reg;

when others => null; -- Silly VHDL.

end case;

-- Increment counter
if (new_outcode_cnt = "101") then
    new_outcode_cnt <= (others => '0');
else
    new_outcode_cnt <= new_outcode_cnt + 1;
end if;

end if;

end if;
end process;

new_z_in_neg(17)      <= not new_z_in(17);
new_z_in_neg(16 downto 0) <= new_z_in(16 downto 0);

-- Component generates Cohen-Sutherland outcode.
new_outcode_gen_0 : outcode_gen
port map (
    reset      => reset,
    clk        => clk,
    x_in       => new_x_in,
    y_in       => new_y_in,

```



```

    z_in      => new_z_in,
    valid_in  => new_valid_in,
    xmin      => new_z_in,
    ymin      => new_z_in,
    zmin      => NEG_ONE,
    xmax      => new_z_in_neg,
    ymax      => new_z_in_neg,
    zmax      => zmax_reg,
    outcode   => new_outcode,
    valid_out => new_outcode_valid
);

-- Delay to match up x,y,z and color with outcode.
new_outcode_xy_dly_prc : process (clk)
begin
    if (clk = '1' and clk'event) then
        new_outcode_xyz_reg(0) <= new_z_in & new_y_in & new_x_in;
        for i in 0 to OUTCODE_LATENCY-2 loop
            new_outcode_xyz_reg(i+1) <= new_outcode_xyz_reg(i);
        end loop;
    end if;
end process;
new_outcode_x    <= new_outcode_xyz_reg(OUTCODE_LATENCY-1)(17 downto 0);
new_outcode_y    <= new_outcode_xyz_reg(OUTCODE_LATENCY-1)(35 downto 18);
new_outcode_z    <= new_outcode_xyz_reg(OUTCODE_LATENCY-1)(53 downto 36);

-- This process cycles through all the calculated outcodes
-- and sends the proper pixel to the arbiter.
pix_sel_prc : process (reset, clk)
begin
    if (reset = '1') then
        fifo_store_done <= '0';
        clip_pix_cnt    <= (others => '0');
        clip_x0         <= (others => '0');
        clip_y0         <= (others => '0');
        clip_z0         <= (others => '0');
        clip_color0     <= (others => '0');
        clip_rdy0       <= '0';
        clip_x1         <= (others => '0');
        clip_y1         <= (others => '0');
        clip_z1         <= (others => '0');
        clip_color1     <= (others => '0');
        clip_rdy1       <= '0';
        clip_rdy        <= '0';
        clip_x0_reg     <= (others => '0');
        clip_y0_reg     <= (others => '0');
        clip_z0_reg     <= (others => '0');
        clip_color0_reg <= (others => '0');
        clip_x1_reg     <= (others => '0');
        clip_y1_reg     <= (others => '0');
        clip_z1_reg     <= (others => '0');
        clip_color1_reg <= (others => '0');
        clip_rdy_reg    <= '0';
    elsif (clk = '1' and clk'event) then

        -- Default
        fifo_store_done <= '0';
        clip_rdy        <= '0';

        if (new_outcode_valid = '1') then

            -- Determine what portion of the volume intersection
            -- we are dealing with. Look at the outcodes and determine what
            -- to do.

```

```

case clip_pix_cnt is
  when "000" =>
    -- Determine if x0,y0,z0 is outside the viewing volume.
    -- If it isn't, use the original coordinates.
    -- If it is see if it was to the left of the volume and if the
new coordinate is
    -- in the volume.
    clip_rdy0 <= '0';
    clip_color0 <= fifo_store_color0;
    if (fifo_store_outcode0 = "00" & x"0") then
      clip_x0 <= fifo_store_x0;
      clip_y0 <= fifo_store_y0;
      clip_z0 <= fifo_store_z0;
      clip_rdy0 <= '1';
    elsif (fifo_store_outcode0(0) = '1' and new_outcode = "00" &
x"0") then
      clip_x0 <= new_outcode_x;
      clip_y0 <= new_outcode_y;
      clip_z0 <= new_outcode_z;
      clip_rdy0 <= '1';
    end if;

    -- Determine if x1,y1,z1 is outside the viewing window.
    -- If it isn't, use the original coordinates.
    -- If it is see if it was to the left of the volume and if the
new coordinate is
    -- in the volume.
    clip_rdy1 <= '0';
    clip_color1 <= fifo_store_color1;
    if (fifo_store_outcode1 = "00" & x"0") then
      clip_x1 <= fifo_store_x1;
      clip_y1 <= fifo_store_y1;
      clip_z1 <= fifo_store_z1;
      clip_rdy1 <= '1';
    elsif (fifo_store_outcode1(0) = '1' and new_outcode = "00" &
x"0") then
      clip_x1 <= new_outcode_x;
      clip_y1 <= new_outcode_y;
      clip_z1 <= new_outcode_z;
      clip_rdy1 <= '1';
    end if;

  when "001" =>
    -- If we haven't already selected the a point, than see if
    -- x0,y0,z0 intersects with the right of the volume.
    if (clip_rdy0 = '0') then
      if (fifo_store_outcode0(1) = '1' and new_outcode = "00" & x"0")
then
        clip_x0 <= new_outcode_x;
        clip_y0 <= new_outcode_y;
        clip_z0 <= new_outcode_z;
        clip_rdy0 <= '1';
      end if;
    end if;

    -- If we haven't already selected the a point, than see if
    -- x1,y1,z1 intersects with the right of the volume.
    if (clip_rdy1 = '0') then
      if (fifo_store_outcode1(1) = '1' and new_outcode = "00" & x"0")
then
        clip_x1 <= new_outcode_x;

```

```

        clip_y1 <= new_outcode_y;
        clip_z1 <= new_outcode_z;
        clip_rdy1 <= '1';
    end if;
end if;

when "010" =>

    -- If we haven't already selected the a point, than see if
    -- x0,y0,z0 intersects with the bottom of the volume.
    if (clip_rdy0 = '0') then
then
        if (fifo_store_outcode0(2) = '1' and new_outcode = "00" & x"0")

            clip_x0 <= new_outcode_x;
            clip_y0 <= new_outcode_y;
            clip_z0 <= new_outcode_z;
            clip_rdy0 <= '1';
        end if;
    end if;

    -- If we haven't already selected the a point, than see if
    -- x1,y1,z1 intersects with the right of the volume.
    if (clip_rdy1 = '0') then
then
        if (fifo_store_outcode1(2) = '1' and new_outcode = "00" & x"0")

            clip_x1 <= new_outcode_x;
            clip_y1 <= new_outcode_y;
            clip_z1 <= new_outcode_z;
            clip_rdy1 <= '1';
        end if;
    end if;

when "011" =>

    -- If we haven't already selected the a point, than see if
    -- x0,y0,z0 intersects with the top of the volume.
    if (clip_rdy0 = '0') then
then
        if (fifo_store_outcode0(3) = '1' and new_outcode = "00" & x"0")

            clip_x0 <= new_outcode_x;
            clip_y0 <= new_outcode_y;
            clip_z0 <= new_outcode_z;
            clip_rdy0 <= '1';
        end if;
    end if;

    -- If we haven't already selected the a point, than see if
    -- x1,y1,z1 intersects with the top of the volume.
    if (clip_rdy1 = '0') then
then
        if (fifo_store_outcode1(3) = '1' and new_outcode = "00" & x"0")

            clip_x1 <= new_outcode_x;
            clip_y1 <= new_outcode_y;
            clip_z1 <= new_outcode_z;
            clip_rdy1 <= '1';
        end if;
    end if;

when "100" =>

    -- If we haven't already selected the a point, than see if
    -- x0,y0,z0 intersects with the back of the volume.
    if (clip_rdy0 = '0') then

```

```

then
    if (fifo_store_outcode0(4) = '1' and new_outcode = "00" & x"0")
        clip_x0 <= new_outcode_x;
        clip_y0 <= new_outcode_y;
        clip_z0 <= new_outcode_z;
        clip_rdy0 <= '1';
    end if;
end if;

-- If we haven't already selected the a point, than see if
-- x1,y1,z1 intersects with the back of the volume.
if (clip_rdy1 = '0') then
then
    if (fifo_store_outcode1(4) = '1' and new_outcode = "00" & x"0")
        clip_x1 <= new_outcode_x;
        clip_y1 <= new_outcode_y;
        clip_z1 <= new_outcode_z;
        clip_rdy1 <= '1';
    end if;
end if;

when "101" =>

    -- If we haven't already selected the a point, than see if
    -- x0,y0,z0 intersects with the front of the volume.
    if (clip_rdy0 = '0') then
then
        if (fifo_store_outcode0(5) = '1' and new_outcode = "00" & x"0")
            clip_x0 <= new_outcode_x;
            clip_y0 <= new_outcode_y;
            clip_z0 <= new_outcode_z;
            clip_rdy0 <= '1';
        end if;
    end if;

    -- If we haven't already selected the a point, than see if
    -- x1,y1,z1 intersects with the front of the volume.
    if (clip_rdy1 = '0') then
then
        if (fifo_store_outcode1(5) = '1' and new_outcode = "00" & x"0")
            clip_x1 <= new_outcode_x;
            clip_y1 <= new_outcode_y;
            clip_z1 <= new_outcode_z;
            clip_rdy1 <= '1';
        end if;
    end if;

    -- Assert clipping coordinates are ready.
    clip_rdy <= '1';
    fifo_store_done <= '1';

    when others => null; -- Silly VHDL.

end case;

-- Increase counter
if (clip_pix_cnt = "101") then
    clip_pix_cnt <= (others => '0');
else
    clip_pix_cnt <= clip_pix_cnt + 1;
end if;

end if;

```

```

-- Register clipping stuff.
if (clip_rdy = '1') then
    clip_rdy_reg    <= '1';
    clip_x0_reg    <= clip_x0;
    clip_y0_reg    <= clip_y0;
    clip_z0_reg    <= clip_z0;
    clip_color0_reg <= clip_color0;
    clip_x1_reg    <= clip_x1;
    clip_y1_reg    <= clip_y1;
    clip_z1_reg    <= clip_z1;
    clip_color1_reg <= clip_color1;
end if;

if (clip_done = '1') then
    clip_rdy_reg <= '0';
end if;

end if;
end process;

-- Arbitration process
-- Selects between clipping and accepted paths.
-- Round robin between the two.
arb_prc : process(reset, clk)
begin
    if (reset = '1') then
        x_out        <= (others => '0');
        y_out        <= (others => '0');
        z_out        <= (others => '0');
        color_out    <= (others => '0');
        valid_out    <= '0';
        clip_done    <= '0';
        fifo_accept_pop <= '0';
        arb_state    <= IDLE;
    elsif (clk = '1' and clk'event) then

        -- Defaults
        valid_out    <= '0';
        clip_done    <= '0';
        fifo_accept_pop <= '0';

        case arb_state is

            when IDLE =>

                if (clip_rdy_reg = '1') then

                    arb_state <= CLIP0;

                elsif (fifo_accept_aempty = '0') then

                    fifo_accept_pop    <= '1';
                    arb_state          <= ACCEPT0;

                end if;

            when CLIP0 =>

                x_out    <= clip_x0_reg;
                y_out    <= clip_y0_reg;
                z_out    <= clip_z0_reg;
                color_out <= clip_color0_reg;
                valid_out <= '1';
                clip_done <= '1';

```

```

    arb_state <= CLIP1;
when CLIP1 =>
    x_out      <= clip_x1_reg;
    y_out      <= clip_y1_reg;
    z_out      <= clip_z1_reg;
    color_out  <= clip_color1_reg;
    valid_out  <= '1';

    if (fifo_accept_aempty = '0') then
        fifo_accept_pop    <= '1';
        arb_state          <= ACCEPT0;
    else
        arb_state <= IDLE;
    end if;
when ACCEPT0 =>
    x_out      <= fifo_accept_rdata(17 downto 0);
    y_out      <= fifo_accept_rdata(35 downto 18);
    z_out      <= fifo_accept_rdata(53 downto 36);
    color_out  <= fifo_accept_rdata(71 downto 54);
    valid_out  <= '1';
    fifo_accept_pop    <= '1';
    arb_state          <= ACCEPT1;

when ACCEPT1 =>
    x_out      <= fifo_accept_rdata(17 downto 0);
    y_out      <= fifo_accept_rdata(35 downto 18);
    z_out      <= fifo_accept_rdata(53 downto 36);
    color_out  <= fifo_accept_rdata(71 downto 54);
    valid_out  <= '1';

    if (clip_rdy_reg = '1') then
        arb_state <= CLIP0;
    else
        arb_state <= IDLE;
    end if;

end case;
end if;
end process;
end hdl;

```

C.8 ABSOLUTE VALUE

This VHDL file calculates the absolute value of the subtraction of two 18bit floating point values.

```

-----
--
--
-- Filename      : abs_val.vhd
--
-- Initial Date  : Feburary 15 2008
--
-- Author       : James Ryan Warner
--
-- Description   : Takes in two integer and calculates their absolutle value.
--
-----
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library work;
use work.gpu_pkg.all;

entity abs_val is

    generic (
        WIDTH : integer := 8
    );

    port (

        -- Reset/Clock
        reset      : in std_logic;
        clk        : in std_logic;

        -- Line Inputs.
        a          : in std_logic_vector(bit_width(WIDTH)-1 downto 0);
        b          : in std_logic_vector(bit_width(WIDTH)-1 downto 0);
        ready      : in std_logic;

        -- Output value
        valid      : out std_logic;
        data       : out std_logic_vector(bit_width(WIDTH)-1 downto 0)

    );

end abs_val;

architecture rtl of abs_val is

    signal mux_sel : std_logic;
    signal asubb   : std_logic_vector(bit_width(WIDTH)-1 downto 0);
    signal bsuba   : std_logic_vector(bit_width(WIDTH)-1 downto 0);

begin

    -- Mux select
    abs_val_prc : process(a, b, ready)
    begin

        -- Determine which value is greater.
        if (a > b) then
            mux_sel <= '1';
        else

```

```

    mux_sel <= '0';
end if;

-- Subtract both values regardless.
asubb <= a - b;
bsuba <= b - a;

end process;

reg_prc : process(clk, reset)
begin
    if (reset = '1') then
        valid <= '0';
        data <= (others => '0');
    elsif (clk'event and clk='1') then
        valid <= ready;
        -- A > B
        if (mux_sel = '1') then
            data <= asubb;
        else
            data <= bsuba;
        end if;
    end if;
end process;

end rtl;

```

C.9 BRESENHAM'S ALGORITHM

This VHDL file is a hardware implementation of Bresenham's line algorithm. It takes two points and interpolates the nearest pixels to activate based on the straight line between the line's endpoints.

```

-----
--
--
-- Filename      : line_drawer.vhd
--
-- Initial Date : January 14 2008
--
-- Author       : James Ryan Warner
--
-- Description  : This block implements a the Bresenham's line algorithm.
--                It takes as input, two 2D points within a given range.
--                It then draws a rasterized line by writting information to
--                the frame buffer. It is assumed that the incomming pixels
are
--                clipped to the viewing volume.
--
--

```



```

-----
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library work;
use work.gpu_pkg.all;

entity line_drawer is

    generic (
        X_PIX_WIDTH : integer := 320;
        Y_PIX_WIDTH : integer := 240
    );

    port (

        -- Reset/Clock
        reset      : in std_logic;
        clk        : in std_logic;

        -- Line Inputs.
        x0         : in std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
        x1         : in std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
        y0         : in std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
        y1         : in std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
        color      : in std_logic_vector(17 downto 0);
        pix_valid  : in std_logic;
        pix_ready  : out std_logic;

        -- Control
        background : in std_logic_vector(35 downto 0);
        enable     : in std_logic;
        eof        : in std_logic;

        -- Memory Outputs
        gpu_req    : out std_logic;
        gpu_rnw    : out std_logic;
        gpu_afull  : in std_logic;
        gpu_addr   : out std_logic_vector(18 downto 0);
        gpu_wpush  : out std_logic;
        gpu_wdata  : out std_logic_vector(35 downto 0);
        gpu_wafull : in std_logic

    );

end line_drawer;

architecture rtl of line_drawer is

    component abs_val
        generic (
            WIDTH : integer
        );
        port (
            -- Reset/Clock
            reset      : in std_logic;
            clk        : in std_logic;

            -- Line Inputs.
            a          : in std_logic_vector(bit_width(WIDTH) - 1 downto 0);

```

```

    b          : in std_logic_vector(bit_width(WIDTH) - 1 downto 0);
    ready      : in std_logic;

    -- Output value
    valid      : out std_logic;
    data       : out std_logic_vector(bit_width(WIDTH) - 1 downto 0)
);
end component;

component fifo_1clk is
    generic (
        FIFO_WIDTH      : integer;
        FIFO_DEPTH      : integer;
        FIFO_AFULL_THRESH : integer;
        FIFO_AEMPTY_THRESH : integer;
        FIFO_FALL_THROUGH : integer
    );
    port (
        -- Clock and reset
        reset : in std_logic;
        clk   : in std_logic;

        -- Control signals
        push  : in std_logic;
        pop   : in std_logic;

        -- Read write data
        wdata : in std_logic_vector(FIFO_WIDTH-1 downto 0);
        rdata : out std_logic_vector(FIFO_WIDTH-1 downto 0);

        -- Status flags.
        afull : out std_logic;
        aempty : out std_logic;
        empty : out std_logic;
        full  : out std_logic
    );
end component;

constant NUM_PIX : integer := X_PIX_WIDTH*Y_PIX_WIDTH;
constant FIFO_WIDTH : integer := bit_width(X_PIX_WIDTH)*4+bit_width(Y_PIX_WIDTH)*3+1+18+2+2+1;

type scan_state_t is
(IDLE, WAIT_FOR_SLOT, CLEAR_SCREEN, DRAW_FIRST, WAIT_DATA, GRAB_DATA, DRAW_LINE);
signal scan_state : scan_state_t;
signal gpu_addr_i : std_logic_vector(16 downto 0);
signal background_reg : std_logic_vector(35 downto 0);
signal eof_dly : std_logic;

-- Fifo Signals.
signal fifo_push : std_logic;
signal fifo_pop : std_logic;
signal fifo_wdata : std_logic_vector(FIFO_WIDTH-1 downto 0);
signal fifo_rdata : std_logic_vector(FIFO_WIDTH-1 downto 0);
signal fifo_afull : std_logic;
signal fifo_empty : std_logic;

-- State 1 signals.
signal s1_valid : std_logic;
signal s1_x0 : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);

```

```

signal s1_x1      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s1_y0      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s1_y1      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s1_color   : std_logic_vector(17 downto 0);

```

-- Stage 2 signals.

```

signal s2_valid   : std_logic;
signal s2_x1x0_abs : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s2_y1y0_abs : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s2_x0      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s2_x1      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s2_y0      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s2_y1      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s2_color   : std_logic_vector(17 downto 0);

```

-- State 3 signals.

```

signal s3_valid   : std_logic;
signal s3_x0      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s3_x1      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s3_y0      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s3_y1      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s3_delta_x : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s3_delta_y : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s3_color   : std_logic_vector(17 downto 0);
signal s3_steep   : std_logic;

```

-- State 4 signals.

```

signal s4_valid   : std_logic;
signal s4_x0      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s4_x1      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s4_y0      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s4_y1      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s4_delta_x : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s4_delta_y : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s4_color   : std_logic_vector(17 downto 0);
signal s4_steep   : std_logic;

```

-- State 5 signals.

```

signal s5_valid   : std_logic;
signal s5_x0      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s5_x1      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s5_y0      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s5_y1      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s5_delta_x : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s5_delta_y : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s5_steep   : std_logic;
signal s5_color   : std_logic_vector(17 downto 0);

```

-- Stage 6 signals

```

signal s6_error   : std_logic_vector(bit_width(X_PIX_WIDTH) downto 0);
signal s6_x0      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s6_x1      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s6_y0      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s6_y1      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s6_valid   : std_logic;
signal s6_delta_x : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal s6_delta_y : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal s6_steep   : std_logic;
signal s6_color   : std_logic_vector(17 downto 0);
signal s6_ystep   : std_logic_vector(1 downto 0);
signal s6_xstep   : std_logic_vector(1 downto 0);

```

```

signal x_pos      : std_logic_vector(bit_width(X_PIX_WIDTH) downto 0);
signal y_pos      : std_logic_vector(bit_width(Y_PIX_WIDTH) downto 0);

```

```

signal x_min      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal y_min      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal x_max      : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal y_max      : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal steep      : std_logic;
signal ystep      : std_logic_vector(1 downto 0);
signal xstep      : std_logic_vector(1 downto 0);
signal line_error : std_logic_vector(bit_width(X_PIX_WIDTH) downto 0);
signal deltax     : std_logic_vector(bit_width(X_PIX_WIDTH) - 1 downto 0);
signal deltay     : std_logic_vector(bit_width(Y_PIX_WIDTH) - 1 downto 0);
signal pix_color  : std_logic_vector(17 downto 0);

```

begin

-- Make sure background only changes on an end of frame.

```
reg_bg_prc : process(clk, reset)
```

```
begin
```

```

  if (reset = '1') then
    background_reg <= (others => '0');
    eof_dly        <= '0';
  elsif (clk'event and clk = '1') then
    eof_dly <= eof;
    if (eof = '1') then
      background_reg <= background;
    end if;
  end if;
end if;
end process;

```

-- Register the data.

-- Clipping should be done, but clamp do it just to be safe.

```
stage1_prc : process (clk, reset)
```

```
begin
```

```

  if (reset = '1') then
    s1_valid <= '0';
    s1_y0    <= (others => '0');
    s1_y1    <= (others => '0');
    s1_x0    <= (others => '0');
    s1_x1    <= (others => '0');
    s1_color <= (others => '0');
  elsif (clk'event and clk = '1') then
    s1_valid <= '0';
    if (pix_valid = '1') then

```

```

      s1_valid <= pix_valid;

```

-- Clamp to max resolution.

```
if (s1_y1 >= Y_PIX_WIDTH) then
```

```

  s1_y1 <= conv_std_logic_vector(Y_PIX_WIDTH - 1, bit_width(Y_PIX_WIDTH));
else

```

```

  s1_y1 <= y1;
end if;
```

```
if (s1_y0 >= Y_PIX_WIDTH) then
```

```

  s1_y0 <= conv_std_logic_vector(Y_PIX_WIDTH - 1, bit_width(Y_PIX_WIDTH));
else

```

```

  s1_y0 <= y0;
end if;
```

```
if (s1_x1 >= X_PIX_WIDTH) then
```

```

  s1_x1 <= conv_std_logic_vector(X_PIX_WIDTH - 1, bit_width(X_PIX_WIDTH));
else

```

```

  s1_x1 <= x1;
end if;
```

```
if (s1_x0 >= X_PIX_WIDTH) then
```

```

  s1_x0 <= conv_std_logic_vector(X_PIX_WIDTH - 1, bit_width(X_PIX_WIDTH));
else

```

```

  s1_x0 <= x0;
end if;
end process;
end;

```

```

        s1_x0 <= x0;
    end if;

    s1_color <= color;

    end if;
end if;
end process;

-- Stage 2 of line drawing pipeline.
-- Determines if the absolute value of x1-x0 and y1-y0
y0y1_abs_val : abs_val
generic map (
    WIDTH => Y_PIX_WIDTH
)
port map(
    reset => reset,
    clk   => clk,
    a     => s1_y0,
    b     => s1_y1,
    ready => s1_valid,
    valid => open,
    data  => s2_y1y0_abs
);

x0x1_abs_val : abs_val
generic map (
    WIDTH => X_PIX_WIDTH
)
port map(
    reset => reset,
    clk   => clk,
    a     => s1_x0,
    b     => s1_x1,
    ready => s1_valid,
    valid => open,
    data  => s2_x1x0_abs
);

-- Stage 2
-- Register all x,y and color info
stage_2_prc : process(clk, reset)
begin
    if (reset = '1') then
        s2_x0    <= (others => '0');
        s2_x1    <= (others => '0');
        s2_y0    <= (others => '0');
        s2_y1    <= (others => '0');
        s2_color <= (others => '0');
        s2_valid <= '0';
    elsif (clk='1' and clk'event) then
        s2_valid <= '0';
        if (s1_valid = '1') then
            s2_valid <= s1_valid;
            s2_x0    <= s1_x0;
            s2_x1    <= s1_x1;
            s2_y0    <= s1_y0;
            s2_y1    <= s1_y1;
            s2_color <= s1_color;
        end if;
    end if;
end process;

```

```
-- Stage 3
-- Determine the steepness of the slope (>1) by comparing y1y0 abs an
-- x1x0 abs. If the slope is steep swap x and y.
stage_3_prc : process(clk, reset)
```

```
begin
  if (reset = '1') then
    s3_x0    <= (others => '0');
    s3_x1    <= (others => '0');
    s3_y0    <= (others => '0');
    s3_y1    <= (others => '0');
    s3_color <= (others => '0');
    s3_steep <= '0';
    s3_valid <= '0';
    s3_deltay <= (others => '0');
    s3_deltax <= (others => '0');
  elsif (clk='1' and clk'event) then
    s3_valid <= '0';
    if (s2_valid = '1') then
      s3_valid <= s2_valid;
      if (s2_y1y0_abs > s2_x1x0_abs) then
        s3_steep <= '1';
      else
        s3_steep <= '0';
      end if;
    end if;
    s3_x0    <= s2_x0;
    s3_x1    <= s2_x1;
    s3_y0    <= s2_y0;
    s3_y1    <= s2_y1;
    s3_color <= s2_color;
    s3_deltay <= s2_y1y0_abs;
    s3_deltax <= s2_x1x0_abs;
  end if;
end if;
end process;
```

```
-- Stage 4
-- Swap x and y order if line is going from right to left.
stage_4_prc : process(clk, reset)
```

```
begin
  if (reset = '1') then
    s4_x0    <= (others => '0');
    s4_x1    <= (others => '0');
    s4_y0    <= (others => '0');
    s4_y1    <= (others => '0');
    s4_color <= (others => '0');
    s4_steep <= '0';
    s4_valid <= '0';
    s4_deltay <= (others => '0');
    s4_deltax <= (others => '0');
  elsif (clk='1' and clk'event) then
    s4_valid <= '0';
    if (s3_valid = '1') then
      s4_valid <= s3_valid;
      if (s3_x0 > s3_x1) then
        s4_x0 <= s3_x1;
        s4_x1 <= s3_x0;
        s4_y0 <= s3_y1;
        s4_y1 <= s3_y0;
      else
        s4_x0 <= s3_x0;
        s4_x1 <= s3_x1;
        s4_y0 <= s3_y0;
        s4_y1 <= s3_y1;
      end if;
    end if;
  end if;
end process;
```

```

        s4_color  <= s3_color;
        s4_steep  <= s3_steep;
        s4_deltax <= s3_deltax;
        s4_deltay <= s3_deltay;
    end if;
end if;
end process;

-- Stage 5
-- Calculate delta x and delta y
stage_5_prc : process(clk, reset)
begin
    if (reset = '1') then
        s5_valid  <= '0';
        s5_steep  <= '0';
        s5_color  <= (others => '0');
        s5_x0     <= (others => '0');
        s5_x1     <= (others => '0');
        s5_y0     <= (others => '0');
        s5_y1     <= (others => '0');
        s5_deltax <= (others => '0');
        s5_deltay <= (others => '0');
    elsif (clk='1' and clk'event) then
        s5_valid <= '0';
        if (s4_valid = '1') then
            s5_valid <= s4_valid;
            s5_color <= s4_color;
            s5_steep <= s4_steep;
            s5_x0    <= s4_x0;
            s5_x1    <= s4_x1;
            s5_y0    <= s4_y0;
            s5_y1    <= s4_y1;
            s5_deltax <= s4_deltax;
            s5_deltay <= s4_deltay;
        end if;
    end if;
end process;

-- Stage 6
-- Calculate error
stage_6_prc : process(clk, reset)
    variable s6_error_v : std_logic_vector(bit_width(X_PIX_WIDTH) downto 0);
begin
    if (reset = '1') then
        s6_error  <= (others => '0');
        s6_valid  <= '0';
        s6_color  <= (others => '0');
        s6_deltax <= (others => '0');
        s6_deltay <= (others => '0');
        s6_steep  <= '0';
        s6_ystep  <= (others => '0');
        s6_xstep  <= (others => '0');
        s6_x0     <= (others => '0');
        s6_x1     <= (others => '0');
        s6_y0     <= (others => '0');
        s6_y1     <= (others => '0');
    elsif (clk='1' and clk'event) then
        s6_valid <= '0';
        if (s5_valid = '1') then
            s6_valid <= s5_valid;
            if (s5_steep = '0') then
                s6_error_v := ("0" & s5_deltax) + 1;
                s6_error_v := "0" & s6_error_v(bit_width(X_PIX_WIDTH) downto 1);
            end if;
        end if;
    end if;
end process;

```

```

        s6_error_v := 0 - s6_error_v;
        s6_error   <= s6_error_v;
    else
        s6_error_v := ("0" & s5_deltax) + 1;
        s6_error_v := "0" & s6_error_v(bit_width(X_PIX_WIDTH) downto 1);
        s6_error_v := 0 - s6_error_v;
        s6_error   <= s6_error_v;
    end if;
    s6_color   <= s5_color;
    s6_deltax  <= s5_deltax;
    s6_deltay  <= s5_deltay;
    s6_steep   <= s5_steep;
    s6_x0      <= s5_x0;
    s6_x1      <= s5_x1;
    s6_y0      <= s5_y0;
    s6_y1      <= s5_y1;
    if (s5_y0 < s5_y1) then
        s6_ystep <= "01";
    elsif (s5_y0 > s5_y1) then
        s6_ystep <= "00";
    else
        s6_ystep <= "11";
    end if;
    if (s5_x0 < s5_x1) then
        s6_xstep <= "01";
    elsif (s5_x0 > s5_x1) then
        s6_xstep <= "00";
    else
        s6_xstep <= "11";
    end if;
end if;
end if;
end process;

-- Wire up fifo signals
fifo_push <= s6_valid;
fifo_wdata <= s6_error & s6_deltax & s6_deltay & s6_steep & s6_ystep &
s6_xstep & s6_color & s6_x0 & s6_x1 & s6_y0 & s6_y1;

-- Fifo data points.
line_fifo : fifo_1clk

generic map (
    FIFO_WIDTH      => FIFO_WIDTH,
    FIFO_DEPTH      => 32,
    FIFO_AFULL_THRESH => 16,
    FIFO_AEMPTY_THRESH=> 0,
    FIFO_FALL_THROUGH => 0
)

port map (
    reset => reset,
    clk   => clk,
    push  => fifo_push,
    pop   => fifo_pop,
    wdata => fifo_wdata,
    rdata => fifo_rdata,
    afull => fifo_afull,
    aempty => open,
    empty => fifo_empty,
    full  => open
);

-- It first recieves and eof for where it clears out the screen with

```



```

-- the background color. Then lines are rasterized until eof.
scan_state_prc : process(clk, reset)
    variable line_error_temp : std_logic_vector(bit_width(X_PIX_WIDTH) downto
0);
begin
    if (reset = '1') then

        -- Reset defaults.
        gpu_req    <= '0';
        gpu_rnw    <= '1';
        gpu_addr_i <= (others => '0');
        gpu_wpush  <= '0';
        gpu_wdata  <= (others => '0');
        scan_state <= IDLE;
        x_pos      <= (others => '0');
        y_pos      <= (others => '0');
        x_min      <= (others => '0');
        y_min      <= (others => '0');
        x_max      <= (others => '0');
        y_max      <= (others => '0');
        steep      <= '0';
        ystep      <= (others => '0');
        xstep      <= (others => '0');
        line_error <= (others => '0');
        deltax     <= (others => '0');
        deltay     <= (others => '0');
        pix_color  <= (others => '0');
        fifo_pop   <= '0';

    elsif (clk = '1' and clk'event) then

        -- Defaults
        gpu_req    <= '0';
        gpu_rnw    <= '1';
        gpu_wpush  <= '0';
        fifo_pop   <= '0';

        -- Raster state machine.
        case scan_state is

            when IDLE =>

                if (eof_dly = '1' and enable = '1') then

                    if (gpu_afull = '0' and gpu_wafull = '0') then

                        -- Goto screen clear state.
                        scan_state <= CLEAR_SCREEN;

                        -- Clear first pixel.
                        gpu_req    <= '1';
                        gpu_rnw    <= '0';
                        gpu_wpush  <= '1';
                        gpu_addr_i <= (others => '0');
                        gpu_wdata  <= background_reg;

                    else

                        -- Wait for memory to become available.
                        scan_state <= WAIT_FOR_SLOT;

                    end if;

                end if;
            end case;
        end process;
    end begin;
end process;

```

```

end if;

when WAIT_FOR_SLOT =>
    if (gpu_afull = '0' and gpu_wafull = '0') then
        -- Goto screen clear state.
        scan_state <= CLEAR_SCREEN;

        -- Clear first pixel.
        gpu_req    <= '1';
        gpu_rnw    <= '0';
        gpu_wpush  <= '1';
        gpu_addr_i <= (others => '0');
        gpu_wdata  <= background_reg;

    end if;

when CLEAR_SCREEN =>
    if (gpu_addr_i = NUM_PIX-1) then
        -- Goto raster state
        scan_state <= DRAW_FIRST;
    else
        if (gpu_afull = '0' and gpu_wafull = '0') then
            -- Clear first pixel.
            gpu_req    <= '1';
            gpu_rnw    <= '0';
            gpu_wpush  <= '1';
            gpu_addr_i <= gpu_addr_i + 1;
            gpu_wdata  <= background_reg;

        end if;

    end if;

when DRAW_FIRST =>
    if (eof_dly = '1') then
        scan_state <= WAIT_FOR_SLOT;
    end if;

    if (gpu_afull = '0' and gpu_wafull = '0' and fifo_empty = '0')
then
        fifo_pop    <= '1';
        scan_state <= WAIT_DATA;

    end if;

when WAIT_DATA =>
    scan_state <= GRAB_DATA;

when GRAB_DATA =>
    x_pos          <= "0" & fifo_rdata((bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2 - 1) downto (bit_width(X_PIX_WIDTH) +
bit_width(Y_PIX_WIDTH)*2));

```

```

y_pos      <= "0" & fifo_rdata((bit_width(Y_PIX_WIDTH)*2-1) downto
(bit_width(Y_PIX_WIDTH)));
y_max      <= fifo_rdata((bit_width(Y_PIX_WIDTH)-1) downto 0);
y_min      <= fifo_rdata((bit_width(Y_PIX_WIDTH)*2-1) downto
(bit_width(Y_PIX_WIDTH)));
x_max      <= fifo_rdata((bit_width(X_PIX_WIDTH) +
bit_width(Y_PIX_WIDTH)*2-1) downto (bit_width(Y_PIX_WIDTH)*2));
x_min      <= fifo_rdata((bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2 - 1) downto (bit_width(X_PIX_WIDTH) +
bit_width(Y_PIX_WIDTH)*2));
pix_color  <= fifo_rdata((18 + bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2 - 1) downto (bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2));
xstep      <= fifo_rdata((2 + 18 + bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2 - 1) downto (18 + bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2));
ystep      <= fifo_rdata((2 + 2 + 18 + bit_width(X_PIX_WIDTH)*2
+ bit_width(Y_PIX_WIDTH)*2 - 1) downto (2 + 18 + bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2));
steep      <= fifo_rdata(2 + 2 + 18 + bit_width(X_PIX_WIDTH)*2 +
bit_width(Y_PIX_WIDTH)*2);
deltay     <= fifo_rdata((2 + 2 + 1 + 18 +
bit_width(X_PIX_WIDTH)*2 + bit_width(Y_PIX_WIDTH)*3 - 1) downto (2 + 2 + 1 +
18 + bit_width(X_PIX_WIDTH)*2 + bit_width(Y_PIX_WIDTH)*2));
deltax     <= fifo_rdata((2 + 2 + 1 + 18 +
bit_width(X_PIX_WIDTH)*3 + bit_width(Y_PIX_WIDTH)*3 - 1) downto (2 + 2 + 1 +
18 + bit_width(X_PIX_WIDTH)*2 + bit_width(Y_PIX_WIDTH)*3));
line_error <= fifo_rdata((2 + 2 + 1 + 18 +
bit_width(X_PIX_WIDTH)*4 + bit_width(Y_PIX_WIDTH)*3 - 2) downto (2 + 2 + 1 +
18 + bit_width(X_PIX_WIDTH)*3 + bit_width(Y_PIX_WIDTH)*3));
scan_state <= DRAW_LINE;

```

```
when DRAW_LINE =>
```

```

  if (eof_dly = '1') then
    scan_state <= WAIT_FOR_SLOT;
  end if;

```

```
  if (gpu_afull = '0' and gpu_wafull = '0') then
```

```

    -- Write a pixel.
    gpu_req   <= '1';
    gpu_rnw   <= '0';
    gpu_wpush <= '1';
    gpu_wdata <= x"0000" & "00" & pix_color;

```

```
    if (steep = '1') then
```

```

      if (((y_max > y_min) and (y_pos(bit_width(Y_PIX_WIDTH)-1) downto
0) <= y_max) and (y_pos(bit_width(Y_PIX_WIDTH)) /= '1')) or
        ((y_max < y_min) and (y_pos(bit_width(Y_PIX_WIDTH)-1) downto
0) >= y_max) and (y_pos(bit_width(Y_PIX_WIDTH)) /= '1')) or
        ((y_max = y_min) and (y_pos(bit_width(Y_PIX_WIDTH)-1) downto
0) = y_max) and (y_pos(bit_width(Y_PIX_WIDTH)) /= '1'))) then

```

```

        gpu_addr_i <=
conv_std_logic_vector((conv_integer(y_pos(bit_width(Y_PIX_WIDTH)-1) downto
0)*X_PIX_WIDTH+conv_integer(x_pos(bit_width(X_PIX_WIDTH)-1) downto 0)), 17);

```

```
        -- Calculate new error.
```

```
        if (line_error(bit_width(X_PIX_WIDTH)) = '1') then
```

```
          line_error_temp := (not line_error) + 1;
```

```
          line_error_temp := ("0" & deltax) - line_error_temp;
```

```
        else
```

```

        line_error_temp := line_error + ("0" & deltax);
    end if;
    if (line_error_temp(bit_width(X_PIX_WIDTH)) = '0') then
        line_error_temp := line_error_temp - ("0" & deltax);
        -- Increment x
        if (xstep = "00") then
            x_pos <= x_pos - 1;
        elsif (xstep = "01") then
            x_pos <= x_pos + 1;
        end if;
    end if;
    line_error <= line_error_temp;

    -- Increment y
    if (y_max > y_min) then
        y_pos <= y_pos + 1;
    else
        y_pos <= y_pos - 1;
    end if;

else
    if (gpu_afull = '0' and gpu_wafull = '0' and fifo_empty =
'0') then
        fifo_pop <= '1';
        scan_state <= WAIT_DATA;
    else
        scan_state <= DRAW_FIRST;
    end if;

end if;

else
    if (((x_max > x_min) and (x_pos(bit_width(X_PIX_WIDTH)) - 1 downto
0) <= x_max) and (x_pos(bit_width(X_PIX_WIDTH)) /= '1')) or
((x_max < x_min) and (x_pos(bit_width(X_PIX_WIDTH)) - 1 downto
0) >= x_max) and (x_pos(bit_width(X_PIX_WIDTH)) /= '1')) or
((x_max = x_min) and (x_pos(bit_width(X_PIX_WIDTH)) - 1 downto
0) = x_max) and (x_pos(bit_width(X_PIX_WIDTH)) /= '1')) then
        gpu_addr_i
conv_std_logic_vector((conv_integer(y_pos(bit_width(Y_PIX_WIDTH)) - 1
0) * X_PIX_WIDTH + conv_integer(x_pos(bit_width(X_PIX_WIDTH)) - 1 downto
0)), 17);
        -- Calculate new error.
        if (line_error(bit_width(X_PIX_WIDTH)) = '1') then
            line_error_temp := (not line_error) + 1;
            line_error_temp := ("0" & deltax) - line_error_temp;
        else
            line_error_temp := line_error + ("0" & deltax);
        end if;
        if (line_error_temp(bit_width(X_PIX_WIDTH)) = '0') then
            line_error_temp := line_error_temp - ("0" & deltax);
            -- Increment y
            if (ystep = "00") then
                y_pos <= y_pos - 1;
            elsif (ystep = "01") then
                y_pos <= y_pos + 1;
            end if;
        end if;
        line_error <= line_error_temp;
    end if;
end if;

```

```

        -- Increment x
        if (x_max > x_min) then
            x_pos <= x_pos + 1;
        else
            x_pos <= x_pos - 1;
        end if;

    else

        if (gpu_afull = '0' and gpu_wafull = '0' and fifo_empty =
'0') then
            fifo_pop <= '1';
            scan_state <= WAIT_DATA;
        else
            scan_state <= DRAW_FIRST;
        end if;

        end if;

    end if;

    end if;

    end case;

end if;

end process;

gpu_addr <= gpu_addr_i & "00";
pix_ready <= not fifo_afull;

end rtl;

```

C.10 ZBT FRAME BUFFER

This VHDL file implements a double buffer. Data from the graphics pipeline is written into one frame buffer while the vga interfaces reads data from another. Each frame switches which buffer the graphics pipeline and vga interface uses. There is also a debug port that gives the CPU raw access to the ZBT memory for debug.

```

-----
--
--
-- Filename      : zbt_frame_intf.vhd
--
-- Initial Data : Oct 23 2007
--

```

```

-- Author      : James Ryan Warner
--
-- Description : This implements the frame buffer.
--              It handles arbitration between 3 ports.
--              1. VGA Read Only port used by the vga controller to
--                 output the latest screen.
--              2. GPU Write interface which interfaces to the
rasterization
--                 and zbuffering logic to update the next frame.
--              3. CPU interface for software updating of the frame buffer.
--              Priority is strict priority with 1 being the highest and
--              3 being the lowest.
--
--              Double buffering is also handled here by using a simple
--              toggeling bit from the vga controllers end of frame.
-----

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

library work;
use work.zbt_ctrl_pkg.all;

```

```

entity zbt_frame_intf is

```

```

    generic (
        ADDR_WIDTH      : integer := 20;
        BYTE_WIDTH      : integer := 9;
        DATA_WIDTH     : integer := 36
    );
    port (
        -- Reset/Clock
        reset           : in std_logic; -- Async Reset.
        sys_clk         : in std_logic; -- System clock.
        zbt_clk         : in std_logic; -- ZBT memory clock.
        vga_clk         : in std_logic; -- Vga clock.

        -- New frame trigger signals.
        gpu_enable      : in std_logic;
        vga_eof         : in std_logic;

        -- VGA Read Only Port
        vga_req         : in std_logic;
        vga_afull       : out std_logic;
        vga_addr        : in std_logic_vector(ADDR_WIDTH-2 downto 0);
        vga_rpop        : in std_logic;
        vga_rdata       : out std_logic_vector(DATA_WIDTH-1 downto 0);
        vga_reempty     : out std_logic;
        vga_rafull      : out std_logic;

        -- GPU Interface port
        gpu_req         : in std_logic;
        gpu_afull       : out std_logic;
        gpu_size        : in std_logic_vector(1 downto 0);
        gpu_addr        : in std_logic_vector(ADDR_WIDTH-2 downto 0);
        gpu_rnw         : in std_logic;
        gpu_wpush       : in std_logic;
        gpu_wdata       : in std_logic_vector(DATA_WIDTH-1 downto 0);
        gpu_wafull      : out std_logic;
        gpu_rpop        : in std_logic;
        gpu_rdata       : out std_logic_vector(DATA_WIDTH-1 downto 0);
        gpu_rwdaddr     : out std_logic_vector(1 downto 0);
        gpu_reempty     : out std_logic;

```

```

-- CPU Interface port.
cpu_sel      : in  std_logic;
cpu_we       : in  std_logic;
cpu_addr     : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
cpu_wdata    : in  std_logic_vector(DATA_WIDTH-1 downto 0);
cpu_wdone    : out std_logic;
cpu_dval     : out std_logic;
cpu_rdata    : out std_logic_vector(DATA_WIDTH-1 downto 0);

-- ZBT interface
zbt_cen      : out std_logic;
zbt_wen      : out std_logic;
zbt_oen      : out std_logic;
zbt_ts       : out std_logic;
zbt_wdata    : out std_logic_vector(DATA_WIDTH-1 downto 0);
zbt_addr     : out std_logic_vector(ADDR_WIDTH-
bit_width(DATA_WIDTH/BYTE_WIDTH)-1 downto 0);
zbt_rdata    : in  std_logic_vector(DATA_WIDTH-1 downto 0)

);

end zbt_frame_intf;

architecture rtl of zbt_frame_intf is

    component zbt_ctrl_top is

        generic (
            NUM_PORTS      : integer;
            ADDR_WIDTH     : integer;
            MEMORY_WIDTH   : integer;
            BYTE_WIDTH     : integer;
            DATA_DELAY    : integer;
            DATA_WIDTH    : integer_array_t;
            MAX_BURST_SIZE : integer_array_t;
            CMD_FIFO_DEPTH : integer_array_t;
            WRITE_FIFO_DEPTH : integer_array_t;
            READ_FIFO_DEPTH : integer_array_t;
            CMD_FIFO_AFULL_THRESH : integer_array_t;
            WRITE_FIFO_AFULL_THRESH : integer_array_t;
            READ_FIFO_AFULL_THRESH : integer_array_t;
            FIFO_DUAL_CLOCK : integer_array_t
        );

        port (
            reset      : in  std_logic;
            clk        : in  std_logic;
            port_clk   : in  std_logic_vector(NUM_PORTS-1 downto 0) := (others =>
'0');

            -- Port interfaces into memory controller.
            port_req   : in  std_logic_vector(NUM_PORTS-1 downto 0);
            port_afull : out std_logic_vector(NUM_PORTS-1 downto 0);
            port_size  : in  integer_array_t;
            std_logic_vector(total_size(conv_array_bit_width(MAX_BURST_SIZE, NUM_PORTS), NUM_PORTS)-1 downto 0);
            port_addr  : in  std_logic_vector(NUM_PORTS*ADDR_WIDTH-1 downto 0);
            port_rnw   : in  std_logic_vector(NUM_PORTS-1 downto 0);
            port_wpush : in  std_logic_vector(NUM_PORTS-1 downto 0);
            port_wdata : in  std_logic_vector((total_size(DATA_WIDTH, NUM_PORTS))-1
downto 0);
            port_wafull : out std_logic_vector(NUM_PORTS-1 downto 0);
            port_rpop  : in  std_logic_vector(NUM_PORTS-1 downto 0);

```

```

    port_rdata : out std_logic_vector((total_size(DATA_WIDTH, NUM_PORTS)) - 1
downto 0);
    port_rwdaddr:
std_logic_vector(total_size(conv_array_bit_width(MAX_BURST_SIZE, NUM_PORTS), NU
M_PORTS) - 1 downto 0);
    port_rempy : out std_logic_vector(NUM_PORTS - 1 downto 0);
    port_rafull : out std_logic_vector(NUM_PORTS - 1 downto 0);

    -- ZBT interface
    zbt_cen      : out std_logic;
    zbt_wen      : out std_logic;
    zbt_ts       : out std_logic;
    zbt_oen      : out std_logic;
    zbt_wdata    : out std_logic_vector(MEMORY_WIDTH - 1 downto 0);
    zbt_addr     : out std_logic_vector(ADDR_WIDTH -
bit_width(MEMORY_WIDTH/BYTE_WIDTH) - 1 downto 0);
    zbt_rdata    : in  std_logic_vector(MEMORY_WIDTH - 1 downto 0)
);

end component;

signal port_clk      : std_logic_vector(2 downto 0);
signal port_req      : std_logic_vector(2 downto 0);
signal port_afull    : std_logic_vector(2 downto 0);
signal port_size     : std_logic_vector(3 downto 0);
signal port_addr     : std_logic_vector(ADDR_WIDTH*3 - 1 downto 0);
signal port_rnw      : std_logic_vector(2 downto 0);
signal port_wpush    : std_logic_vector(2 downto 0);
signal port_wdata    : std_logic_vector(DATA_WIDTH*3 - 1 downto 0);
signal port_wafull   : std_logic_vector(2 downto 0);
signal port_rpop     : std_logic_vector(2 downto 0);
signal port_rdata    : std_logic_vector(DATA_WIDTH*3 - 1 downto 0);
signal port_rwdaddr  : std_logic_vector(3 downto 0);
signal port_rempy    : std_logic_vector(2 downto 0);
signal port_rafull   : std_logic_vector(2 downto 0);

signal wframe_addr   : std_logic;
signal rframe_addr   : std_logic;
type eof_state_t is (IDLE, ENABLE_GPU, ENABLE_CPU);
signal eof_state     : eof_state_t;

signal cpu_mem_req   : std_logic;
signal cpu_mem_afull : std_logic;
signal cpu_mem_addr  : std_logic_vector(ADDR_WIDTH - 1 downto 0);
signal cpu_mem_rnw   : std_logic;
signal cpu_mem_wpush : std_logic;
signal cpu_mem_wafull : std_logic;
signal cpu_mem_wdata : std_logic_vector(DATA_WIDTH - 1 downto 0);
signal cpu_mem_rpop  : std_logic;
signal cpu_mem_rdata : std_logic_vector(DATA_WIDTH - 1 downto 0);
signal cpu_mem_rempy : std_logic;

type cpu_mbox_state_t is (IDLE, READ_CMD, READ_WAIT, READ_POP, WRITE_CMD);
signal cpu_mbox_state: cpu_mbox_state_t;

```

```
begin
```

```

-- Cpu mailbox interface process.
-- Converts mailbox signals to memory port interface signals.
cpu_mail_prc : process(sys_clk, reset)
begin
    if (reset = '1') then

```



```

cpu_wdone      <= '0';
cpu_dval       <= '0';
cpu_mem_req    <= '0';
cpu_mem_addr   <= (others => '0');
cpu_mem_rnw    <= '1';
cpu_mem_wpush  <= '0';
cpu_mem_wdata  <= (others => '0');
cpu_mem_rpop   <= '0';
cpu_mbox_state <= IDLE;

elsif (sys_clk = '1' and sys_clk'event) then

  -- Defaults.
  cpu_wdone      <= '0';
  cpu_dval       <= '0';
  cpu_mem_req    <= '0';
  cpu_mem_rnw    <= '1';
  cpu_mem_wpush  <= '0';
  cpu_mem_rpop   <= '0';

  -- Control state machine.
  case cpu_mbox_state is

    when IDLE =>

      -- Wait for CPU to trigger a read or write.
      if (cpu_sel = '1') then
        if (cpu_we = '0') then

          cpu_mem_req    <= '1';
          cpu_mem_rnw    <= '1';
          cpu_mem_addr   <= cpu_addr;
          cpu_mbox_state <= READ_CMD;

        else

          cpu_mem_req    <= '1';
          cpu_mem_rnw    <= '0';
          cpu_mem_wpush  <= '1';
          cpu_mem_addr   <= cpu_addr;
          cpu_mem_wdata  <= cpu_wdata;
          cpu_mbox_state <= WRITE_CMD;

        end if;
      end if;

    when READ_CMD =>

      if (cpu_mem_reempty = '1') then
        cpu_mbox_state <= READ_WAIT;
      else
        cpu_mem_rpop    <= '1';
        cpu_mbox_state <= READ_POP;
      end if;

    when READ_WAIT =>

      if (cpu_mem_reempty = '0') then
        cpu_mem_rpop    <= '1';
        cpu_mbox_state <= READ_POP;
      end if;

    when READ_POP =>

```

```

    cpu_dval      <= '1';
    cpu_mbox_state <= IDLE;

when WRITE_CMD =>

    cpu_wdone     <= '1';
    cpu_mbox_state <= IDLE;

end case;

end if;
end process;
cpu_rdata <= cpu_mem_rdata;

-- Process used to toggle double frame buffer.
-- as well as detect end of frame.
det_eof_prc : process (sys_clk, reset)
begin
    if (reset = '1') then

        wframe_addr <= '0';
        rframe_addr <= '0';
        eof_state    <= IDLE;

    elsif (sys_clk = '1' and sys_clk'event) then

        case eof_state is

            when IDLE =>
                if (gpu_enable = '1') then
                    if (vga_eof = '1') then
                        wframe_addr <= '0';
                        rframe_addr <= '1';
                        eof_state    <= ENABLE_GPU;
                    end if;
                else
                    if (vga_eof = '1') then
                        wframe_addr <= '0';
                        rframe_addr <= '0';
                        eof_state    <= ENABLE_CPU;
                    end if;
                end if;

            when ENABLE_GPU =>
                if (gpu_enable = '1') then
                    if (vga_eof = '1') then
                        wframe_addr <= not wframe_addr;
                        rframe_addr <= not rframe_addr;
                    end if;
                else
                    if (vga_eof = '1') then
                        wframe_addr <= '0';
                        rframe_addr <= '0';
                        eof_state    <= ENABLE_CPU;
                    end if;
                end if;

            when ENABLE_CPU =>
                if (gpu_enable = '1') then
                    if (vga_eof = '1') then
                        wframe_addr <= '0';
                        rframe_addr <= '1';
                        eof_state    <= ENABLE_GPU;
                    end if;
                end if;
            end case;
        end if;
    end process;

```

```

        else
        if (vga_eof = '1') then
            wframe_addr <= '0';
            rframe_addr <= '0';
        end if;
    end if;
end case;
end if;
end process;

port_clk    <= sys_clk    & sys_clk                & vga_clk;
port_req    <= cpu_mem_req & gpu_req                & vga_req;
port_size   <= '0'      & gpu_size                & '0';
port_addr   <= cpu_mem_addr & wframe_addr & gpu_addr & rframe_addr &
vga_addr;
port_rnw    <= cpu_mem_rnw & gpu_rnw                & '1';
port_wpush  <= cpu_mem_wpush & gpu_wpush            & '0';
port_wdata  <= cpu_mem_wdata & gpu_wdata            & x"00000000";
port_rpop   <= cpu_mem_rpop & gpu_rpop              & vga_rpop;

cpu_mem_afull <= port_afull(2);
gpu_afull    <= port_afull(1);
vga_afull    <= port_afull(0);
cpu_mem_wafull <= port_wafull(2);
gpu_wafull   <= port_wafull(1);
cpu_mem_rdata <= port_rdata(DATA_WIDTH*3-1 downto DATA_WIDTH*2);
gpu_rdata    <= port_rdata(DATA_WIDTH*2-1 downto DATA_WIDTH*1);
vga_rdata    <= port_rdata(DATA_WIDTH*1-1 downto DATA_WIDTH*0);
gpu_rwdaddr  <= port_rwdaddr(2 downto 1);
cpu_mem_reempty <= port_reempty(2);
gpu_reempty  <= port_reempty(1);
vga_reempty  <= port_reempty(0);
vga_rafull   <= port_rafull(0);

zbt_ctrl_top_inst : zbt_ctrl_top
generic map (
    NUM_PORTS          => 3,
    ADDR_WIDTH         => ADDR_WIDTH,
    DATA_WIDTH        => (DATA_WIDTH, DATA_WIDTH, DATA_WIDTH, 0),
    MEMORY_WIDTH       => DATA_WIDTH,
    BYTE_WIDTH         => 9,
    DATA_DELAY        => 2,
    MAX_BURST_SIZE     => (1, 4, 1, 0),
    CMD_FIFO_DEPTH     => (16, 16, 16, 0),
    WRITE_FIFO_DEPTH   => (64, 64, 64, 0),
    READ_FIFO_DEPTH    => (64, 64, 64, 0),
    CMD_FIFO_AFULL_THRESH => (8, 8, 8, 0),
    WRITE_FIFO_AFULL_THRESH => (32, 32, 32, 0),
    READ_FIFO_AFULL_THRESH => (32, 32, 32, 0),
    FIFO_DUAL_CLOCK    => (1, 0, 0, 0)
)
port map (
    reset      => reset,
    clk        => zbt_clk,
    port_clk   => port_clk,
    port_req   => port_req,
    port_afull => port_afull,
    port_size  => port_size,
    port_addr  => port_addr,
    port_rnw   => port_rnw,
    port_wpush => port_wpush,
    port_wdata => port_wdata,
    port_wafull => port_wafull,
    port_rpop  => port_rpop,

```

```

port_rdata    => port_rdata,
port_rwdaddr => port_rwdaddr,
port_empty   => port_empty,
port_rafull  => port_rafull,
zbt_cen      => zbt_cen,
zbt_wen      => zbt_wen,
zbt_oen      => zbt_oen,
zbt_ts       => zbt_ts,
zbt_wdata    => zbt_wdata,
zbt_addr     => zbt_addr,
zbt_rdata    => zbt_rdata
);
end rtl;

```

C.11 ZBT MEMORY CONTROLLER

This VHDL file provides a multiport interface to the ZBT memory. It provides a configurable number of ports which are arbitrated in fair round robin fashion. Three ports are defined in this design. A read only VGA port, a write only graphics pipeline port, and a bidirectional CPU debug port.

```

-----
--
--
-- Filename      : zbt_ctrl_top.vhd
--
-- Initial Data  : May 23 2007
--
-- Author       : James Ryan Warner
--
-- Description   : This block provides a multiport interface to a static zbt
--                 memory. The arbitration is done in a round robin fashion.
--                 The number of ports as well as data and address widths are
--                 configurable.
-----
--

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

library work;
use work.zbt_ctrl_pkg.all;

```

```

entity zbt_ctrl_top is

```

```

    generic (
        NUM_PORTS      : integer := MAX_NUM_INTF;
        ADDR_WIDTH     : integer := DEFAULT_ADDR_WIDTH;
        MEMORY_WIDTH    : integer := DEFAULT_MEMORY_WIDTH;
    );
end entity zbt_ctrl_top;

```

```

    BYTE_WIDTH           : integer := DEFAULT_BYTE_WIDTH;
    DATA_DELAY          : integer := DEFAULT_DATA_DELAY;
    DATA_WIDTH          : integer_array_t := DEFAULT_DATA_WIDTH;
    MAX_BURST_SIZE       : integer_array_t := DEFAULT_MAX_BURST_SIZE;
    CMD_FIFO_DEPTH       : integer_array_t := DEFAULT_CMD_FIFO_DEPTH;
    WRITE_FIFO_DEPTH     : integer_array_t := DEFAULT_WRITE_FIFO_DEPTH;
    READ_FIFO_DEPTH      : integer_array_t := DEFAULT_READ_FIFO_DEPTH;
    CMD_FIFO_AFULL_THRESH : integer_array_t :=
DEFAULT_CMD_FIFO_AFULL_THRESH;
    WRITE_FIFO_AFULL_THRESH : integer_array_t :=
DEFAULT_WRITE_FIFO_AFULL_THRESH;
    READ_FIFO_AFULL_THRESH : integer_array_t :=
DEFAULT_READ_FIFO_AFULL_THRESH;
    FIFO_DUAL_CLOCK      : integer_array_t := DEFAULT_FIFO_DUAL_CLOCK
);

port (
-- Reset/Clock
reset      : in std_logic; -- Async Reset.
clk        : in std_logic; -- Memory clock
port_clk   : in std_logic_vector(NUM_PORTS-1 downto 0) := (others =>
'0');

-- Port interfaces into memory controller.
port_req   : in std_logic_vector(NUM_PORTS-1 downto 0);
port_afull : out std_logic_vector(NUM_PORTS-1 downto 0);
port_size  : in
std_logic_vector(total_size(conv_array_bit_width(MAX_BURST_SIZE, NUM_PORTS), NUM_PORTS)-1 downto 0);
port_addr  : in std_logic_vector(NUM_PORTS*ADDR_WIDTH-1 downto 0);
port_rnw   : in std_logic_vector(NUM_PORTS-1 downto 0);
port_wpush : in std_logic_vector(NUM_PORTS-1 downto 0);
port_wdata : in std_logic_vector((total_size(DATA_WIDTH, NUM_PORTS))-1
downto 0);
port_wafull : out std_logic_vector(NUM_PORTS-1 downto 0);
port_rpop  : in std_logic_vector(NUM_PORTS-1 downto 0);
port_rdata : out std_logic_vector((total_size(DATA_WIDTH, NUM_PORTS))-1
downto 0);
port_rwaddr: out
std_logic_vector(total_size(conv_array_bit_width(MAX_BURST_SIZE, NUM_PORTS), NUM_PORTS)-1 downto 0);
port_remap : out std_logic_vector(NUM_PORTS-1 downto 0);
port_rafull : out std_logic_vector(NUM_PORTS-1 downto 0);

-- ZBT interface
zbt_cen    : out std_logic;
zbt_wen    : out std_logic;
zbt_oen    : out std_logic;
zbt_ts     : out std_logic;
zbt_wdata  : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
zbt_addr   : out std_logic_vector(ADDR_WIDTH-1
bit_width(MEMORY_WIDTH/BYTE_WIDTH)-1 downto 0);
zbt_rdata  : in std_logic_vector(MEMORY_WIDTH-1 downto 0)

);

end zbt_ctrl_top;

architecture rtl of zbt_ctrl_top is
    constant MAX_MEM_BURST_SIZE : integer := max_size(MAX_BURST_SIZE, NUM_PORTS)
* (max_size(DATA_WIDTH, NUM_PORTS)/MEMORY_WIDTH);
    constant MAX_MEM_BURST_BITS : integer := bit_width(MAX_MEM_BURST_SIZE);
    constant NUM_PORT_BITS      : integer := bit_width(NUM_PORTS);

```

```

constant TOTAL_DATA_WIDTH : integer := total_size(DATA_WIDTH, NUM_PORTS);
constant TOTAL_ADDR_WIDTH : integer := NUM_PORTS * ADDR_WIDTH;
constant TOTAL_BURST_WIDTH : integer :=
total_size(conv_array_bit_width(MAX_BURST_SIZE, NUM_PORTS), NUM_PORTS);
constant MAX_BURST_BITS : integer_array_t :=
conv_array_bit_width(MAX_BURST_SIZE, NUM_PORTS);

```

```

component zbt_port_interface is

```

```

generic (

```

```

ADDR_WIDTH : integer;
DATA_WIDTH : integer;
MAX_BURST_SIZE : integer;
CMD_FIFO_DEPTH : integer;
WRITE_FIFO_DEPTH : integer;
READ_FIFO_DEPTH : integer;
CMD_FIFO_AFULL_THRESH : integer;
WRITE_FIFO_AFULL_THRESH : integer;
READ_FIFO_AFULL_THRESH : integer;
FIFO_DUAL_CLOCK : integer

```

```

);

```

```

port (

```

```

reset : in std_logic; -- Async Reset.
clk : in std_logic; -- Memory clock
port_clk : in std_logic;
port_req : in std_logic;
port_afull : out std_logic;
port_size : in std_logic_vector(bit_width(MAX_BURST_SIZE) - 1
downto 0);
port_addr : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
port_rnw : in std_logic;
port_wpush : in std_logic;
port_wdata : in std_logic_vector(DATA_WIDTH - 1 downto 0);
port_wafull : out std_logic;
port_rpop : in std_logic;
port_rdata : out std_logic_vector(DATA_WIDTH - 1 downto 0);
port_rwaddr : out std_logic_vector(bit_width(MAX_BURST_SIZE) - 1
downto 0);
port_reempty : out std_logic;
port_rafull : out std_logic;
arb_req : out std_logic;
arb_ack : in std_logic;
arb_size : out std_logic_vector(bit_width(MAX_BURST_SIZE) - 1
downto 0);
arb_rnw : out std_logic;
arb_addr : out std_logic_vector(ADDR_WIDTH - 1 downto 0);
arb_wdata : out std_logic_vector(DATA_WIDTH - 1 downto 0);
arb_wdata_empty : out std_logic;
arb_wdata_pop : in std_logic;
arb_rdata : in std_logic_vector(DATA_WIDTH - 1 downto 0);
arb_dval : in std_logic;
arb_rwaddr : in std_logic_vector(bit_width(MAX_BURST_SIZE) - 1
downto 0)
);

```

```

end component;

```

```

component zbt_width_conversion is

```

```

generic (

```

```

ADDR_WIDTH : integer;
DATA_WIDTH : integer;
MEMORY_WIDTH : integer;
BYTE_WIDTH : integer;
MAX_BURST_SIZE : integer;
MAX_MEM_BURST_SIZE : integer

```

```

);

```

```

port (
  reset      : in std_logic;      -- Async Reset.
  clk       : in std_logic;      -- Memory clock
  arb_size_i : in std_logic_vector(bit_width(MAX_BURST_SIZE) - 1
downto 0);
  arb_addr_i : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
  arb_size_o : out std_logic_vector(bit_width(MAX_MEM_BURST_SIZE) - 1
downto 0);
  arb_addr_o : out std_logic_vector(ADDR_WIDTH - 1 downto 0);
  arb_wdata_i : in std_logic_vector(DATA_WIDTH - 1 downto 0);
  arb_wdata_pop_i : in std_logic;
  arb_wdata_o : out std_logic_vector(MEMORY_WIDTH - 1 downto 0);
  arb_wdata_pop_o : out std_logic;
  arb_rdata_i : in std_logic_vector(MEMORY_WIDTH - 1 downto 0);
  arb_dval_i  : in std_logic;
  arb_rwdaddr_i : in std_logic_vector(bit_width(MAX_MEM_BURST_SIZE) - 1
downto 0);
  arb_rdata_o  : out std_logic_vector(DATA_WIDTH - 1 downto 0);
  arb_dval_o   : out std_logic;
  arb_rwdaddr_o : out std_logic_vector(bit_width(MAX_BURST_SIZE) - 1
downto 0);
);
end component;

component zbt_port_arbiter is
  generic (
    NUM_PORTS      : integer;
    ADDR_WIDTH     : integer;
    MEMORY_WIDTH   : integer;
    MAX_BURST_SIZE : integer
  );
  port (
    reset      : in std_logic;      -- Async Reset.
    clk       : in std_logic;      -- Memory clock
    req       : in std_logic_vector(NUM_PORTS - 1 downto 0);
    ack       : out std_logic_vector(NUM_PORTS - 1 downto 0);
    size      : in
std_logic_vector(NUM_PORTS * bit_width(MAX_MEM_BURST_SIZE) - 1 downto 0);
    rnw       : in std_logic_vector(NUM_PORTS - 1 downto 0);
    addr      : in std_logic_vector(NUM_PORTS * ADDR_WIDTH - 1 downto 0);
    wdata     : in std_logic_vector(NUM_PORTS * MEMORY_WIDTH - 1 downto 0);
    wdata_empty : in std_logic_vector(NUM_PORTS - 1 downto 0);
    wdata_pop : out std_logic_vector(NUM_PORTS - 1 downto 0);
    rdata     : out std_logic_vector(NUM_PORTS * MEMORY_WIDTH - 1 downto 0);
    rval      : out std_logic_vector(NUM_PORTS - 1 downto 0);
    rwdaddr   : out
std_logic_vector(NUM_PORTS * bit_width(MAX_MEM_BURST_SIZE) - 1 downto 0);
    mem_sel   : out std_logic;
    mem_rnw   : out std_logic;
    mem_addr  : out std_logic_vector(ADDR_WIDTH - 1 downto 0);
    mem_wdata : out std_logic_vector(MEMORY_WIDTH - 1 downto 0);
    mem_wdaddr : out
std_logic_vector(bit_width(MAX_MEM_BURST_SIZE) - 1
downto 0);
    mem_wport : out std_logic_vector(bit_width(NUM_PORTS) - 1 downto 0);
    mem_dval  : in std_logic;
    mem_rdata : in std_logic_vector(MEMORY_WIDTH - 1 downto 0);
    mem_rport : in std_logic_vector(bit_width(NUM_PORTS) - 1 downto 0);
    mem_rwdaddr : in
std_logic_vector(bit_width(MAX_MEM_BURST_SIZE) - 1
downto 0);
  );
end component;

component zbt_intf is
  generic (

```

```

ADDR_WIDTH      : integer;
MEMORY_WIDTH    : integer;
DATA_DELAY      : integer;
NUM_PORTS       : integer;
MAX_BURST_SIZE : integer
);

port (
  reset      : in std_logic; -- Async Reset.
  clk       : in std_logic; -- Memory clock
  mem_sel    : in std_logic;
  mem_rnw    : in std_logic;
  mem_addr   : in std_logic_vector(ADDR_WIDTH-1 downto 0);
  mem_wdata  : in std_logic_vector(MEMORY_WIDTH-1 downto 0);
  mem_wdaddr : in std_logic_vector(bit_width(MAX_MEM_BURST_SIZE)-1
downto 0);
  mem_wport  : in std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
  mem_dval   : out std_logic;
  mem_rdata  : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
  mem_rport  : out std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
  mem_rwdaddr : out std_logic_vector(bit_width(MAX_MEM_BURST_SIZE)-1
downto 0);
  zbt_cen    : out std_logic;
  zbt_wen    : out std_logic;
  zbt_oen    : out std_logic;
  zbt_ts     : out std_logic;
  zbt_wdata  : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
  zbt_addr   : out std_logic_vector(ADDR_WIDTH-1 downto 0);
  zbt_rdata  : in std_logic_vector(MEMORY_WIDTH-1 downto 0)
);
end component;

signal conv_size      : std_logic_vector(TOTAL_BURST_WIDTH-1 downto 0);
signal conv_addr     : std_logic_vector(TOTAL_ADDR_WIDTH-1 downto 0);
signal conv_wdata    : std_logic_vector(TOTAL_DATA_WIDTH-1 downto 0);
signal conv_wdata_pop : std_logic_vector(NUM_PORTS-1 downto 0);
signal conv_rdata    : std_logic_vector(TOTAL_DATA_WIDTH-1 downto 0);
signal conv_dval     : std_logic_vector(NUM_PORTS-1 downto 0);
signal conv_rwdaddr  : std_logic_vector(TOTAL_BURST_WIDTH-1 downto 0);

signal arb_req       : std_logic_vector(NUM_PORTS-1 downto 0);
signal arb_ack      : std_logic_vector(NUM_PORTS-1 downto 0);
signal arb_size     : std_logic_vector(NUM_PORTS*MAX_MEM_BURST_BITS-1
downto 0);
signal arb_rnw      : std_logic_vector(NUM_PORTS-1 downto 0);
signal arb_addr     : std_logic_vector(NUM_PORTS*ADDR_WIDTH-1 downto 0);
signal arb_wdata    : std_logic_vector(NUM_PORTS*MEMORY_WIDTH-1 downto
0);
signal arb_wdata_empty : std_logic_vector(NUM_PORTS-1 downto 0);
signal arb_wdata_pop : std_logic_vector(NUM_PORTS-1 downto 0);
signal arb_rdata    : std_logic_vector(NUM_PORTS*MEMORY_WIDTH-1 downto
0);
signal arb_dval     : std_logic_vector(NUM_PORTS-1 downto 0);
signal arb_rwdaddr  : std_logic_vector(NUM_PORTS*MAX_MEM_BURST_BITS-1
downto 0);

signal mem_sel      : std_logic;
signal mem_rnw     : std_logic;
signal mem_addr    : std_logic_vector(ADDR_WIDTH-1 downto 0);
signal mem_wdata   : std_logic_vector(MEMORY_WIDTH-1 downto 0);
signal mem_wdaddr  : std_logic_vector(MAX_MEM_BURST_BITS-1 downto 0);
signal mem_wport   : std_logic_vector(NUM_PORT_BITS-1 downto 0);
signal mem_dval    : std_logic;

```



```

signal mem_rdata      : std_logic_vector(MEMORY_WIDTH-1 downto 0);
signal mem_rport      : std_logic_vector(NUM_PORT_BITS-1 downto 0);
signal mem_rwaddr     : std_logic_vector(MAX_MEM_BURST_BITS-1 downto 0);

begin

port_interface_gen : for i in 0 to NUM_PORTS-1 generate
begin
    port_interface_inst : zbt_port_interface
        generic map (
            ADDR_WIDTH           => ADDR_WIDTH,
            DATA_WIDTH          => DATA_WIDTH(i),
            MAX_BURST_SIZE      => MAX_BURST_SIZE(i),
            CMD_FIFO_DEPTH      => CMD_FIFO_DEPTH(i),
            WRITE_FIFO_DEPTH    => WRITE_FIFO_DEPTH(i),
            READ_FIFO_DEPTH     => READ_FIFO_DEPTH(i),
            CMD_FIFO_AFULL_THRESH => CMD_FIFO_AFULL_THRESH(i),
            WRITE_FIFO_AFULL_THRESH => WRITE_FIFO_AFULL_THRESH(i),
            READ_FIFO_AFULL_THRESH => READ_FIFO_AFULL_THRESH(i),
            FIFO_DUAL_CLOCK     => FIFO_DUAL_CLOCK(i)
        )
        port map (
            reset                => reset,
            clk                  => clk,
            port_clk             => port_clk(i),
            port_req             => port_req(i),
            port_afull          => port_afull(i),
            port_size            => port_size(top_index(i, MAX_BURST_BITS)
downto bottom_index(i, MAX_BURST_BITS)),
            port_addr            => port_addr(top_index(i, ADDR_WIDTH) downto
bottom_index(i, ADDR_WIDTH)),
            port_rnw             => port_rnw(i),
            port_wpush          => port_wpush(i),
            port_wdata          => port_wdata(top_index(i, DATA_WIDTH) downto
bottom_index(i, DATA_WIDTH)),
            port_wafull         => port_wafull(i),
            port_rpop           => port_rpop(i),
            port_rdata          => port_rdata(top_index(i, DATA_WIDTH) downto
bottom_index(i, DATA_WIDTH)),
            port_rwaddr         => port_rwaddr(top_index(i, MAX_BURST_BITS)
downto bottom_index(i, MAX_BURST_BITS)),
            port_remap          => port_remap(i),
            port_rafull         => port_rafull(i),
            arb_req             => arb_req(i),
            arb_ack             => arb_ack(i),
            arb_size            => conv_size(top_index(i, MAX_BURST_BITS)
downto bottom_index(i, MAX_BURST_BITS)),
            arb_rnw             => arb_rnw(i),
            arb_addr            => conv_addr(top_index(i, ADDR_WIDTH) downto
bottom_index(i, ADDR_WIDTH)),
            arb_wdata          => conv_wdata(top_index(i, DATA_WIDTH) downto
bottom_index(i, DATA_WIDTH)),
            arb_wdata_empty    => arb_wdata_empty(i),
            arb_wdata_pop      => conv_wdata_pop(i),
            arb_rdata          => conv_rdata(top_index(i, DATA_WIDTH) downto
bottom_index(i, DATA_WIDTH)),
            arb_dval            => conv_dval(i),
            arb_rwaddr         => conv_rwaddr(top_index(i, MAX_BURST_BITS)
downto bottom_index(i, MAX_BURST_BITS))
        );

    zbt_width_conversion_inst : zbt_width_conversion
        generic map (

```

```

        ADDR_WIDTH      => ADDR_WIDTH,
        DATA_WIDTH     => DATA_WIDTH(i),
        MEMORY_WIDTH    => MEMORY_WIDTH,
        BYTE_WIDTH      => BYTE_WIDTH,
        MAX_BURST_SIZE  => MAX_BURST_SIZE(i),
        MAX_MEM_BURST_SIZE => MAX_MEM_BURST_SIZE
    )
    port map (
        reset           => reset,
        clk             => clk,
        arb_size_i      => conv_size(top_index(i, MAX_BURST_BITS) downto
bottom_index(i, MAX_BURST_BITS)),
        arb_addr_i      => conv_addr(top_index(i, ADDR_WIDTH) downto
bottom_index(i, ADDR_WIDTH)),
        arb_size_o      => arb_size(top_index(i, MAX_MEM_BURST_BITS) downto
bottom_index(i, MAX_MEM_BURST_BITS)),
        arb_addr_o      => arb_addr(top_index(i, ADDR_WIDTH) downto
bottom_index(i, ADDR_WIDTH)),
        arb_wdata_i     => conv_wdata(top_index(i, DATA_WIDTH) downto
bottom_index(i, DATA_WIDTH)),
        arb_wdata_pop_i => arb_wdata_pop(i),
        arb_wdata_o     => arb_wdata(top_index(i, MEMORY_WIDTH) downto
bottom_index(i, MEMORY_WIDTH)),
        arb_wdata_pop_o => conv_wdata_pop(i),
        arb_rdata_i     => arb_rdata(top_index(i, MEMORY_WIDTH) downto
bottom_index(i, MEMORY_WIDTH)),
        arb_dval_i      => arb_dval(i),
        arb_rwdaddr_i   => arb_rwdaddr(top_index(i, MAX_MEM_BURST_BITS) downto
bottom_index(i, MAX_MEM_BURST_BITS)),
        arb_rdata_o     => conv_rdata(top_index(i, DATA_WIDTH) downto
bottom_index(i, DATA_WIDTH)),
        arb_dval_o      => conv_dval(i),
        arb_rwdaddr_o   => conv_rwdaddr(top_index(i, MAX_BURST_BITS) downto
bottom_index(i, MAX_BURST_BITS))
    );

```

```
end generate port_interface_gen;
```

```

zbt_port_arbiter_inst : zbt_port_arbiter
generic map(
    NUM_PORTS      => NUM_PORTS,
    ADDR_WIDTH     => ADDR_WIDTH,
    MEMORY_WIDTH   => MEMORY_WIDTH,
    MAX_BURST_SIZE => MAX_MEM_BURST_SIZE
)
port map(
    reset           => reset,
    clk             => clk,
    req             => arb_req,
    ack             => arb_ack,
    size           => arb_size,
    rnw             => arb_rnw,
    addr           => arb_addr,
    wdata          => arb_wdata,
    wdata_empty    => arb_wdata_empty,
    wdata_pop      => arb_wdata_pop,
    rdata          => arb_rdata,
    rval           => arb_dval,
    rwdaddr        => arb_rwdaddr,
    mem_sel        => mem_sel,
    mem_rnw        => mem_rnw,
    mem_addr       => mem_addr,
    mem_wdata      => mem_wdata,
    mem_wdaddr     => mem_wdaddr,

```

```

        mem_wport      => mem_wport,
        mem_dval       => mem_dval,
        mem_rdata      => mem_rdata,
        mem_rport      => mem_rport,
        mem_rwdaddr    => mem_rwdaddr
    );

zbt_intf_inst : zbt_intf
generic map (
    ADDR_WIDTH      => (ADDR_WIDTH - bit_width(MEMORY_WIDTH/BYTE_WIDTH)),
    MEMORY_WIDTH    => MEMORY_WIDTH,
    DATA_DELAY     => DATA_DELAY,
    NUM_PORTS       => NUM_PORTS,
    MAX_BURST_SIZE  => MAX_MEM_BURST_SIZE
)
port map(
    reset           => reset,
    clk             => clk,
    mem_sel         => mem_sel,
    mem_rnw         => mem_rnw,
    mem_addr        => mem_addr(ADDR_WIDTH -
bit_width(MEMORY_WIDTH/BYTE_WIDTH) - 1 downto 0),
    mem_wdata       => mem_wdata,
    mem_wdaddr      => mem_wdaddr,
    mem_wport       => mem_wport,
    mem_dval        => mem_dval,
    mem_rdata       => mem_rdata,
    mem_rport       => mem_rport,
    mem_rwdaddr     => mem_rwdaddr,
    zbt_cen         => zbt_cen,
    zbt_wen         => zbt_wen,
    zbt_oen         => zbt_oen,
    zbt_ts          => zbt_ts,
    zbt_wdata       => zbt_wdata,
    zbt_addr        => zbt_addr,
    zbt_rdata       => zbt_rdata
);

end rtl;

```

C.12 ZBT PHYSICAL INTERFACE

This VHDL file interfaces directly to the zbt memory. All the necessary control signals are driven out to memory for either reads or writers.

```

-----
--
--
-- Filename      : zbt_intf.vhd
--
-- Initial Data : May 23 2007
--
-- Author       : James Ryan Warner
--

```

```
-- Description : The output signals are to interface directly with the zbt
memory.
--          Certian featur's of the ZBT memory such as byte enables and
burst
--          mode are disabled as they are not needed for our design.
--          The DCM for output clock generation and tri-state buffers
are
--          assumed to exist at a higher level.
-----
--
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
```

```
library work;
use work.zbt_ctrl_pkg.all;
```

```
entity zbt_intf is
```

```
generic (
  ADDR_WIDTH      : integer;
  MEMORY_WIDTH    : integer;
  DATA_DELAY     : integer;
  NUM_PORTS       : integer;
  MAX_BURST_SIZE  : integer
);
```

```
port (
  -- Reset/Clock
  reset      : in std_logic; -- Async Reset.
  clk        : in std_logic; -- Memory clock

  -- Arbitration interface
  mem_sel    : in std_logic;
  mem_rnw    : in std_logic;
  mem_addr   : in std_logic_vector(ADDR_WIDTH-1 downto 0);
  mem_wdata  : in std_logic_vector(MEMORY_WIDTH-1 downto 0);
  mem_wdaddr : in std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto 0);
  mem_wport  : in std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
  mem_dval   : out std_logic;
  mem_rdata  : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
  mem_rport  : out std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
  mem_rwdaddr : out std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto 0);

  -- ZBT interface
  zbt_cen    : out std_logic;
  zbt_wen    : out std_logic;
  zbt_oen    : out std_logic;
  zbt_ts     : out std_logic;
  zbt_wdata  : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
  zbt_addr   : out std_logic_vector(ADDR_WIDTH-1 downto 0);
  zbt_rdata  : in std_logic_vector(MEMORY_WIDTH-1 downto 0)
);
```

```
end zbt_intf;
```

```
architecture rtl of zbt_intf is
```

```
-- Delay signals.
type delay_data_type is array (0 to DATA_DELAY-1) of
std_logic_vector(MEMORY_WIDTH-1 downto 0);
```

```

type delay_port_type is array (0 to DATA_DELAY) of
std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
type delay_wdaddr_type is array (0 to DATA_DELAY) of
std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto 0);
signal zbt_oen_d : std_logic_vector(DATA_DELAY-1 downto 0);
signal zbt_wdata_d : delay_data_type;
signal delay_port : delay_port_type;
signal delay_wdaddr : delay_wdaddr_type;
signal delay_dval : std_logic_vector(DATA_DELAY downto 0);

```

```
begin
```

```

process (reset, clk)
begin

```

```

    if (reset = '1') then

```

```

        zbt_cen      <= '1';
        zbt_wen      <= '1';
        zbt_addr     <= (others => '0');
        zbt_wdata    <= (others => '0');
        zbt_oen      <= '1';
        zbt_ts       <= '0';

```

```

        mem_dval     <= '0';
        mem_rdata    <= (others => '0');
        mem_rport    <= (others => '0');
        mem_rwaddr   <= (others => '0');

```

```

        for i in 0 to DATA_DELAY-1 loop
            zbt_oen_d(i) <= '1';
            zbt_wdata_d(i) <= (others => '0');
        end loop;

```

```

        for i in 0 to DATA_DELAY loop
            delay_dval(i) <= '0';
            delay_port(i) <= (others => '0');
            delay_wdaddr(i) <= (others => '0');
        end loop;

```

```

    elsif (clk = '1' and clk'event) then

```

```

        zbt_cen      <= '1';
        zbt_wen      <= '1';
        zbt_oen_d(0) <= '1';
        delay_dval(0) <= '0';
        zbt_addr     <= (others => '0');
        zbt_wdata_d(0) <= (others => '0');
        if (mem_sel = '1' and mem_rnw = '0') then
            zbt_cen      <= '0';
            zbt_wen      <= '0';
            zbt_wdata_d(0) <= mem_wdata;
            zbt_addr     <= mem_addr + mem_wdaddr;
            delay_port(0) <= mem_wport;
            delay_wdaddr(0) <= mem_wdaddr;
            elsif (mem_sel = '1' and mem_rnw = '1') then
                zbt_cen      <= '0';
                zbt_addr     <= mem_addr + mem_wdaddr;
            delay_dval(0) <= '1';
            zbt_oen_d(0) <= '0';
            delay_port(0) <= mem_wport;
            delay_wdaddr(0) <= mem_wdaddr;
        end if;

```

```

for i in 1 to DATA_DELAY-1 loop
  zbt_oen_d(i)   <= zbt_oen_d(i-1);
  zbt_wdata_d(i) <= zbt_wdata_d(i-1);
end loop;

for i in 1 to DATA_DELAY loop
  delay_dval(i)  <= delay_dval(i-1);
  delay_port(i)  <= delay_port(i-1);
  delay_wdaddr(i) <= delay_wdaddr(i-1);
end loop;

zbt_oen   <= zbt_oen_d(DATA_DELAY-1);
zbt_wdata <= zbt_wdata_d(DATA_DELAY-1);
zbt_ts    <= not zbt_oen_d(DATA_DELAY-1);

mem_dval  <= delay_dval(DATA_DELAY);
mem_rport <= delay_port(DATA_DELAY);
mem_rwaddr <= delay_wdaddr(DATA_DELAY);
mem_rdata <= zbt_rdata;

end if;

end process;

end rtl;

```

C.13 ZBT PORT INTERFACE

This VHDL file defines the logic for a single port in the zbt memory controller.

```

-----
--
--
-- Filename      : zbt_port_interface.vhd
--
-- Initial Date  : May 23 2007
--
-- Author       : James Ryan Warner
--
-- Description   : This block implements a singular port interface to zbt
--                 memory controller. It consists of 3 fifos. The fifos are
--                 the read, write and command fifos. The write and command
--                 fifos are used to buffer data to be written to zbt memory.
--                 The read fifo buffers data coming from the zbt memory.
--
-----
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library work;
use work.zbt_ctrl_pkg.all;

```

```

entity zbt_port_interface is
  generic (
    ADDR_WIDTH           : integer;
    DATA_WIDTH          : integer;
    MAX_BURST_SIZE      : integer;
    CMD_FIFO_DEPTH      : integer;
    WRITE_FIFO_DEPTH    : integer;
    READ_FIFO_DEPTH     : integer;
    CMD_FIFO_AFULL_THRESH : integer;
    WRITE_FIFO_AFULL_THRESH : integer;
    READ_FIFO_AFULL_THRESH : integer;
    FIFO_DUAL_CLOCK     : integer
  );
  port (
    -- Reset/Clock
    reset      : in std_logic;      -- Async Reset.
    clk       : in std_logic;      -- Memory clock
    port_clk  : in std_logic := '0';

    -- External port interface to command fifo.
    port_req  : in std_logic;
    port_afull : out std_logic;
    port_size : in std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto
0);
    port_addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    port_rnw  : in std_logic;

    -- External port interface to write fifo.
    port_wpush : in std_logic;
    port_wdata : in std_logic_vector(DATA_WIDTH-1 downto 0);
    port_wafull : out std_logic;

    -- External port interface to read fifo
    port_rpop  : in std_logic;
    port_rdata : out std_logic_vector(DATA_WIDTH-1 downto 0);
    port_rwaddr : out std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto
0);
    port_rempty : out std_logic;
    port_rafull : out std_logic;

    -- Arbiter interface to port cmd fifo
    arb_req  : out std_logic;
    arb_ack  : in std_logic;
    arb_size : out std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto
0);
    arb_rnw  : out std_logic;
    arb_addr : out std_logic_vector(ADDR_WIDTH-1 downto 0);

    -- Arbiter interface to write fifo
    arb_wdata : out std_logic_vector(DATA_WIDTH-1 downto 0);
    arb_wdata_empty : out std_logic;
    arb_wdata_pop : in std_logic;

    -- Arbiter interface to port read fifo
    arb_rdata : in std_logic_vector(DATA_WIDTH-1 downto 0);
    arb_dval  : in std_logic;
    arb_rwaddr : in std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto
0);
  );

```

```
end zbt_port_interface;
```

```
architecture rtl of zbt_port_interface is
```

```
-- Various constants for vector widths and fifo widths.
constant NUM_BURST_BITS : integer := bit_width(MAX_BURST_SIZE);
constant CMD_FIFO_WIDTH : integer := 1 + ADDR_WIDTH + NUM_BURST_BITS;
constant CMD_ADDR_START : integer := 0;
constant CMD_ADDR_END : integer := ADDR_WIDTH-1;
constant CMD_SIZE_START : integer := CMD_ADDR_END+1;
constant CMD_SIZE_END : integer := CMD_SIZE_START+NUM_BURST_BITS-1;
constant CMD_RNW_POS : integer := CMD_SIZE_END+1;
constant DATA_FIFO_WIDTH : integer := NUM_BURST_BITS + DATA_WIDTH;
constant DATA_DATA_START : integer := 0;
constant DATA_DATA_END : integer := DATA_WIDTH-1;
constant DATA_WORD_START : integer := DATA_DATA_END+1;
constant DATA_WORD_END : integer := DATA_WORD_START+NUM_BURST_BITS-1;

-- Command fifo data record.
type cmd_fifo_rec_t is record
  rnw : std_logic;
  size : std_logic_vector(NUM_BURST_BITS-1 downto 0);
  addr : std_logic_vector(ADDR_WIDTH-1 downto 0);
end record;

-- Command fifo reset record constant.
constant CMD_FIFO_RECORD_RESET : cmd_fifo_rec_t := (
  rnw => '0',
  size => (others => '0'),
  addr => (others => '0')
);

-- Converts command fifo data record to std_logic_vector
function cmd_fifo_rec_to_slv ( rec : cmd_fifo_rec_t )
  return std_logic_vector is
  variable temp : std_logic_vector(CMD_FIFO_WIDTH-1 downto 0) := (others =>
'0');
begin
  temp := rec.rnw & rec.size & rec.addr;
  return temp;
end cmd_fifo_rec_to_slv;

-- Converts standard logic vector to command fifo data record type.
function slv_to_cmd_fifo_rec ( vec : std_logic_vector(CMD_FIFO_WIDTH-1
downto 0) )
  return cmd_fifo_rec_t is
  variable temp : cmd_fifo_rec_t := CMD_FIFO_RECORD_RESET;
begin
  temp.addr := vec(CMD_ADDR_END downto CMD_ADDR_START);
  temp.size := vec(CMD_SIZE_END downto CMD_SIZE_START);
  temp.rnw := vec(CMD_RNW_POS);
  return temp;
end slv_to_cmd_fifo_rec;

-- Data fifo data record.
type data_fifo_rec_t is record
  word : std_logic_vector(NUM_BURST_BITS-1 downto 0);
  data : std_logic_vector(DATA_WIDTH-1 downto 0);
end record;

-- Data fifo reset record constant.
constant DATA_FIFO_RECORD_RESET : data_fifo_rec_t := (
  word => (others => '0'),
  data => (others => '0')
```



```

);

-- Converts data fifo record to std_logic_vector
function data_fifo_rec_to_slv ( rec : data_fifo_rec_t )
  return std_logic_vector is
  variable temp : std_logic_vector(DATA_FIFO_WIDTH-1 downto 0) := (others
=> '0');
begin
  temp := rec.word & rec.data;
  return temp;
end data_fifo_rec_to_slv;

-- Converts standard logic vector to data fifo record type.
function slv_to_data_fifo_rec ( vec : std_logic_vector(DATA_FIFO_WIDTH-1
downto 0) )
  return data_fifo_rec_t is
  variable temp : data_fifo_rec_t := DATA_FIFO_RECORD_RESET;
begin
  temp.data := vec(DATA_DATA_END downto DATA_DATA_START);
  temp.word := vec(DATA_WORD_END downto DATA_WORD_START);
  return temp;
end slv_to_data_fifo_rec;

component fifo_1clk is
  generic (
    FIFO_WIDTH      : integer := 18;
    FIFO_DEPTH      : integer := 16;
    FIFO_AFULL_THRESH : integer := 8;
    FIFO_FALL_THROUGH : integer := 0
  );
  port (
    reset : in  std_logic;
    clk   : in  std_logic;
    push  : in  std_logic;
    pop   : in  std_logic;
    wdata : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
    rdata : out std_logic_vector(FIFO_WIDTH-1 downto 0);
    afull : out std_logic;
    empty : out std_logic
  );
end component;

component fifo_2clk is
  generic (
    FIFO_WIDTH      : integer := 18;
    FIFO_DEPTH      : integer := 16;
    FIFO_AFULL_THRESH : integer := 8;
    FIFO_FALL_THROUGH : integer := 0
  );
  port (
    reset : in  std_logic;
    wclk  : in  std_logic;
    rclk  : in  std_logic;
    push  : in  std_logic;
    pop   : in  std_logic;
    wdata : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
    rdata : out std_logic_vector(FIFO_WIDTH-1 downto 0);
    afull : out std_logic;
    empty : out std_logic
  );
end component;

-- Command fifo signals.
signal cmd_fifo_push : std_logic;

```

```

signal cmd_fifo_pop      : std_logic;
signal cmd_fifo_wdata   : cmd_fifo_rec_t;
signal cmd_fifo_rdata   : std_logic_vector(CMD_FIFO_WIDTH-1 downto 0);
signal cmd_fifo_afull   : std_logic;
signal cmd_fifo_afull_d : std_logic;
signal cmd_fifo_empty   : std_logic;

-- Write fifo signals.
signal write_fifo_push  : std_logic;
signal write_fifo_pop   : std_logic;
signal write_fifo_wdata : std_logic_vector(DATA_WIDTH-1 downto 0);
signal write_fifo_rdata : std_logic_vector(DATA_WIDTH-1 downto 0);
signal write_fifo_afull : std_logic;
signal write_fifo_empty : std_logic;

-- Read fifo signals.
signal read_fifo_push   : std_logic;
signal read_fifo_pop    : std_logic;
signal read_fifo_wdata  : data_fifo_rec_t;
signal read_fifo_rdata  : std_logic_vector(NUM_BURST_BITS+DATA_WIDTH-1
downto 0);
signal read_fifo_afull  : std_logic;
signal read_fifo_empty  : std_logic;

signal arb_ctrl         : cmd_fifo_rec_t;
signal port_rdata_waddr: data_fifo_rec_t;

begin

single_clock_on : if (FIFO_DUAL_CLOCK = 0) generate

-- Instansate command fifo
cmd_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => CMD_FIFO_WIDTH,
    FIFO_DEPTH      => CMD_FIFO_DEPTH,
    FIFO_AFULL_THRESH => CMD_FIFO_AFULL_THRESH,
    FIFO_FALL_THROUGH => 1
)
port map (
    reset => reset,
    clk   => clk,
    push  => cmd_fifo_push,
    pop   => cmd_fifo_pop,
    wdata => cmd_fifo_rec_to_slv(cmd_fifo_wdata),
    rdata => cmd_fifo_rdata,
    afull => cmd_fifo_afull,
    empty => cmd_fifo_empty
);

-- Used for edge detection.
cmd_afull_falling_edge_prc : process(clk, reset)
begin
    if (reset = '1') then
        cmd_fifo_afull_d <= '0';
    elsif (clk='1' and clk'event) then
        cmd_fifo_afull_d <= cmd_fifo_afull;
    end if;
end process;

-- Instansate write data fifo
write_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => DATA_WIDTH,

```

```

    FIFO_DEPTH      => WRITE_FIFO_DEPTH,
    FIFO_AFULL_THRESH => WRITE_FIFO_AFULL_THRESH,
    FIFO_FALL_THROUGH => 0
)
port map (
    reset => reset,
    clk   => clk,
    push  => write_fifo_push,
    pop   => write_fifo_pop,
    wdata => write_fifo_wdata,
    rdata => write_fifo_rdata,
    afull => write_fifo_afull,
    empty => write_fifo_empty
);

-- Instansate read data fifo
read_fifo : fifo_1clk
generic map (
    FIFO_WIDTH      => DATA_FIFO_WIDTH,
    FIFO_DEPTH      => READ_FIFO_DEPTH,
    FIFO_AFULL_THRESH => READ_FIFO_AFULL_THRESH,
    FIFO_FALL_THROUGH => 0
)
port map (
    reset => reset,
    clk   => clk,
    push  => read_fifo_push,
    pop   => read_fifo_pop,
    wdata => data_fifo_rec_to_slv(read_fifo_wdata),
    rdata => read_fifo_rdata,
    afull => read_fifo_afull,
    empty => read_fifo_empty
);

end generate;

dual_clock_on : if (FIFO_DUAL_CLOCK /= 0) generate

-- Instansate command fifo
cmd_fifo : fifo_2clk
generic map (
    FIFO_WIDTH      => CMD_FIFO_WIDTH,
    FIFO_DEPTH      => CMD_FIFO_DEPTH,
    FIFO_AFULL_THRESH => CMD_FIFO_AFULL_THRESH,
    FIFO_FALL_THROUGH => 1
)
port map (
    reset => reset,
    wclk  => port_clk,
    rclk  => clk,
    push  => cmd_fifo_push,
    pop   => cmd_fifo_pop,
    wdata => cmd_fifo_rec_to_slv(cmd_fifo_wdata),
    rdata => cmd_fifo_rdata,
    afull => cmd_fifo_afull,
    empty => cmd_fifo_empty
);

-- Used for edge dectection.
cmd_afull_falling_edge_prc : process(port_clk, reset)
begin
    if (reset = '1') then
        cmd_fifo_afull_d <= '0';
    elsif (port_clk='1' and port_clk'event) then

```

```

        cmd_fifo_afull_d <= cmd_fifo_afull;
    end if;
end process;

-- Instansate write data fifo
write_fifo : fifo_2clk
generic map (
    FIFO_WIDTH      => DATA_WIDTH,
    FIFO_DEPTH      => WRITE_FIFO_DEPTH,
    FIFO_AFULL_THRESH => WRITE_FIFO_AFULL_THRESH,
    FIFO_FALL_THROUGH => 0
)
port map (
    reset => reset,
    wclk  => port_clk,
    rclk  => clk,
    push  => write_fifo_push,
    pop   => write_fifo_pop,
    wdata => write_fifo_wdata,
    rdata => write_fifo_rdata,
    afull => write_fifo_afull,
    empty => write_fifo_empty
);

-- Instansate read data fifo
read_fifo : fifo_2clk
generic map (
    FIFO_WIDTH      => DATA_FIFO_WIDTH,
    FIFO_DEPTH      => READ_FIFO_DEPTH,
    FIFO_AFULL_THRESH => READ_FIFO_AFULL_THRESH,
    FIFO_FALL_THROUGH => 0
)
port map (
    reset => reset,
    wclk  => clk,
    rclk  => port_clk,
    push  => read_fifo_push,
    pop   => read_fifo_pop,
    wdata => data_fifo_rec_to_slv(read_fifo_wdata),
    rdata => read_fifo_rdata,
    afull => read_fifo_afull,
    empty => read_fifo_empty
);

end generate;

-- wire up command fifo input signals
cmd_fifo_push      <= port_req;
cmd_fifo_pop       <= arb_ack;
cmd_fifo_wdata.rnw <= port_rnw;
cmd_fifo_wdata.size <= port_size;
cmd_fifo_wdata.addr <= port_addr;

-- port ack pulses high asynchronously with port request,
-- so long as fifo is not full.
port_afull <= cmd_fifo_afull;

-- Wire out arb ready signal and data to arbitration block.
arb_req  <= not cmd_fifo_empty;
arb_ctrl <= slv_to_cmd_fifo_rec(cmd_fifo_rdata);
arb_addr <= arb_ctrl.addr;
arb_rnw  <= arb_ctrl.rnw;
arb_size <= arb_ctrl.size;

```

```

-- wire up write fifo input signals
write_fifo_push  <= port_wpush;
write_fifo_pop   <= arb_wdata_pop;
write_fifo_wdata <= port_wdata;

-- wire up write fifo output signals
port_wafull      <= write_fifo_afull;
arb_wdata_empty <= write_fifo_empty;
arb_wdata        <= write_fifo_rdata;

-- wire up command fifo input signals
read_fifo_push   <= arb_dval;
read_fifo_pop    <= port_rpop;
read_fifo_wdata.data <= arb_rdata;
read_fifo_wdata.word <= arb_rwdaddr;

-- Write out read fifo outputs.
port_rdata_wdaddr <= slv_to_data_fifo_rec(read_fifo_rdata);
port_rdata        <= port_rdata_wdaddr.data;
port_rwdaddr      <= port_rdata_wdaddr.word;
port_rempy        <= read_fifo_empty;
port_rafull       <= read_fifo_afull;

end rtl;

```

C.14 ZBT ARBITER

This VHDL file defines the logic for the round robin arbitration of the memory controller's multi-port interface.

```

-----
--
--
-- Filename      : zbt_port_arbiter.vhd
--
-- Initial Date : May 23 2007
--
-- Author       : James Ryan Warner
--
-- Description  : This block implements a simple round robin arbitration
scheme
--               for multi port interface to a zbt memory.  The number of
ports
--               is configurable.
--
-----
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library work;

```

```

use work.zbt_ctrl_pkg.all;

entity zbt_port_arbiter is

    generic (
        NUM_PORTS      : integer;
        ADDR_WIDTH     : integer;
        MEMORY_WIDTH   : integer;
        MAX_BURST_SIZE : integer
    );

    port (

        -- Reset/Clock
        reset      : in  std_logic;          -- Async Reset.
        clk        : in  std_logic;          -- Memory clock

        -- Request/Ack signals
        req        : in  std_logic_vector(NUM_PORTS-1 downto 0);
        ack        : out std_logic_vector(NUM_PORTS-1 downto 0);

        -- Control signals
        size       : in  std_logic_vector(NUM_PORTS*bit_width(MAX_BURST_SIZE)-1
downto 0);
        rnw        : in  std_logic_vector(NUM_PORTS-1 downto 0);
        addr       : in  std_logic_vector(NUM_PORTS*ADDR_WIDTH-1 downto 0);

        -- Write data
        wdata      : in  std_logic_vector(NUM_PORTS*MEMORY_WIDTH-1 downto 0);
        wdata_empty: in  std_logic_vector(NUM_PORTS-1 downto 0);
        wdata_pop  : out std_logic_vector(NUM_PORTS-1 downto 0);

        -- Read data
        rdata      : out std_logic_vector(NUM_PORTS*MEMORY_WIDTH-1 downto 0);
        rval       : out std_logic_vector(NUM_PORTS-1 downto 0);
        rwdaddr    : out std_logic_vector(NUM_PORTS*bit_width(MAX_BURST_SIZE)-1
downto 0);

        -- Memory interface signals
        mem_sel    : out std_logic;
        mem_rnw    : out std_logic;
        mem_addr   : out std_logic_vector(ADDR_WIDTH-1 downto 0);
        mem_wdata  : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
        mem_wdaddr : out std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto 0);
        mem_wport  : out std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
        mem_dval   : in  std_logic;
        mem_rdata  : in  std_logic_vector(MEMORY_WIDTH-1 downto 0);
        mem_rport  : in  std_logic_vector(bit_width(NUM_PORTS)-1 downto 0);
        mem_rwdaddr: in  std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto 0)

    );

end zbt_port_arbiter;

architecture rtl of zbt_port_arbiter is

    constant MAX_BURST_BITS      : integer := bit_width(MAX_BURST_SIZE);
    constant WDADDR_VECTOR_SIZE : integer := NUM_PORTS * MAX_BURST_BITS;
    constant DATA_VECTOR_SIZE  : integer := NUM_PORTS * MEMORY_WIDTH;
    constant ADDR_VECTOR_SIZE   : integer := NUM_PORTS * ADDR_WIDTH;

    type wdaddr_array_t is array (0 to NUM_PORTS-1) of
std_logic_vector(MAX_BURST_BITS-1 downto 0);

```

```

type data_array_t is array (0 to NUM_PORTS-1) of
std_logic_vector(MEMORY_WIDTH-1 downto 0);
type addr_array_t is array (0 to NUM_PORTS-1) of
std_logic_vector(ADDR_WIDTH-1 downto 0);

signal size_s      : wdaddr_array_t;
signal size_s_reg  : wdaddr_array_t;
signal wdata_s     : data_array_t;
signal addr_s      : addr_array_t;
signal rdata_s     : data_array_t;
signal rwdaddr_s   : wdaddr_array_t;
signal rnw_s       : std_logic_vector(NUM_PORTS-1 downto 0);

signal ack_s       : std_logic_vector(NUM_PORTS-1 downto 0);
signal wdata_pop_s : std_logic_vector(NUM_PORTS-1 downto 0);

type arb_state_t is (IDLE, GRANT);
signal state       : arb_state_t;
signal port_state  : integer range 0 to NUM_PORTS-1;
signal arb_count   : std_logic_vector(MAX_BURST_BITS-1 downto 0);

signal rnw_d       : std_logic_vector(NUM_PORTS-1 downto 0);
signal addr_d      : addr_array_t;
signal arb_count_d : std_logic_vector(MAX_BURST_BITS-1 downto 0);
signal state_d     : arb_state_t;
signal port_state_d : integer range 0 to NUM_PORTS-1;
signal ack_d       : std_logic_vector(NUM_PORTS-1 downto 0);

```

```
begin
```

```

-- These signal assignments in the async process are merely for convenience.
-- They convert the raw vector ports to an array of vectors.
-- This makes writing code much easier.

```

```

vector_convert_proc : process (size, addr, wdata, rdata_s, rwdaddr_s)
    variable start_index : integer := 0;
    variable end_index   : integer := 0;
begin

```

```

    for i in 0 to NUM_PORTS-1 loop
        -- Wire up size vector.
        start_index := i * MAX_BURST_BITS;
        end_index   := (i+1) * MAX_BURST_BITS - 1;
        size_s(i)   <= size(end_index downto start_index);
        -- Wire up wdata vector.
        start_index := i * MEMORY_WIDTH;
        end_index   := (i+1) * MEMORY_WIDTH - 1;
        wdata_s(i)  <= wdata(end_index downto start_index);
        -- Wire up rwdaddr vector.
        start_index := i * MAX_BURST_BITS;
        end_index   := (i+1) * MAX_BURST_BITS - 1;
        rwdaddr(end_index downto start_index) <= rwdaddr_s(i);
        -- Wire up rdata vector.
        start_index := i * MEMORY_WIDTH;
        end_index   := (i+1) * MEMORY_WIDTH - 1;
        rdata(end_index downto start_index) <= rdata_s(i);
    end loop;

```

```
end process;
```

```

vector_convert_proc_seq : process (clk, reset)
    variable start_index : integer := 0;
    variable end_index   : integer := 0;
begin
    if (reset = '1') then

```

```

    addr_s <= (others => (others => '0'));
    rnw_s <= (others => '0');
    elsif (clk = '1' and clk'event) then
        for i in 0 to NUM_PORTS-1 loop
            -- Wire up address vector.
            start_index := i * ADDR_WIDTH;
            end_index   := (i+1) * ADDR_WIDTH - 1;
            addr_s(i)   <= addr(end_index downto start_index);
        end loop;
        rnw_s <= rnw;
    end if;
end process;

-- This process handles the round robin arbitration of memory port
accesses.
-- A port gets access for as many cycles as the burst size requests.
arb_state_proc : process(reset, clk)
    variable index      : integer;
    variable index2     : integer;
    variable other_req  : std_logic := '0';
    variable another_req : std_logic := '0';
begin
    if (reset = '1') then
        -- Reset values.
        state      <= IDLE;
        port_state <= 0;
        ack_s      <= (others => '0');
        wdata_pop_s <= (others => '0');
        arb_count  <= (others => '0');
        size_s_reg <= (others => (others => '0'));
    elsif (clk = '1' and clk'event) then
        -- Defaults
        ack_s      <= (others => '0');
        wdata_pop_s <= (others => '0');

        case state is
            when IDLE =>
                -- Go through each port and if request is seen return the proper
                -- ack and goto grant state while setting proper port state.
                for i in 0 to NUM_PORTS-1 loop
                    -- Go through the previous request.
                    -- If any are high then it preempts the current req.
                    other_req := '0';
                    for k in 0 to i-1 loop
                        other_req := req(k) or other_req;
                    end loop;

                    if (req(i) = '1' and other_req = '0') then
                        -- If this request is seen goto grant state and
                        state <= GRANT;

                        -- Set the arb counter to zero
                        -- set the proper port state bit.
                        arb_count <= (others => '0');
                        port_state <= i;

                        -- Assert ack signal.

```



```

ack_s(i) <= '1';

-- Assert write pop when writing.
wdata_pop_s <= (others => '0');
if (rnw(i) = '0') then
    wdata_pop_s(i) <= '1';
else
    wdata_pop_s(i) <= '0';
end if;

-- Register size.
size_s_reg(i) <= size_s(i);

end if;
end loop;

when GRANT =>

-- Go through each port.
for i in 0 to NUM_PORTS-1 loop

    -- Determine other req.
    another_req := '0';
    for j in 0 to NUM_PORTS-1 loop
        if (req(j) = '1' and j /= i) then
            another_req := '1';
        end if;
    end loop;

    -- If this port is active service it.
    if (port_state = i) then

        -- Wait till arb count equals size until releasing port.
        if (arb_count /= size_s_reg(i)) then

            -- Increment arb count and assert wdata_pop on writes only.
            arb_count <= arb_count + 1;

            -- Hold the pop signal, whatever it is.
            wdata_pop_s(i) <= wdata_pop_s(i);

            elsif (size_s_reg(i) = 0 and req(i) = '1' and another_req =
'0') then

                -- Idle cycle required for single cycle transfers.
                state <= IDLE;

            elsif (arb_count = size_s_reg(i) and req /= 0) then

                -- Hold the pop signal, whatever it is.
                wdata_pop_s(i) <= wdata_pop_s(i);

                -- Go through each port, starting at the next port and
                -- ending at the current port.
                for j in 0 to NUM_PORTS-1 loop

                    -- I hope the tool is smart enough not to synthesize this.
                    index := (j + i + 1) mod NUM_PORTS;

                    -- Go through the previous request.
                    -- If any are high then it preempts the current req.
                    other_req := '0';
                    for k in 0 to j-1 loop
                        index2 := (k + i + 1) mod NUM_PORTS;

```

```

        other_req := req(index2) or other_req;
    end loop;

    if (req(index) = '1' and other_req = '0') then

        -- Set the arb counter to zero and
        -- set the proper port state bit.
        arb_count      <= (others => '0');
        port_state     <= index;

        -- Assert ack signal.
        ack_s(index) <= '1';

        -- Assert write pop when writing.
        wdata_pop_s <= (others => '0');
        if (rnw(index) = '0') then
            wdata_pop_s(index) <= '1';
        else
            wdata_pop_s(index) <= '0';
        end if;

        -- Register size.
        size_s_reg(index) <= size_s(index);

    end if;

end loop;

elsif (arb_count = size_s_reg(i) and req = 0) then

    -- No request lines are active, return to idle state.
    state <= IDLE;

end if;

end if;

end loop;

end case;

end if;

end process;

-- Wire out outputs because stupid vhdl won't let you read output ports!!!!
wdata_pop <= wdata_pop_s;
ack       <= ack_s;

memory_drive_prc : process(clk, reset)
begin
    if (reset = '1') then

        -- Do nothing
        mem_sel   <= '0';
        mem_rnw   <= '1';
        mem_addr  <= (others => '0');
        mem_wdata <= (others => '0');
        mem_wport <= (others => '0');
        mem_wdaddr <= (others => '0');

        rnw_d     <= (others => '0');
        addr_d    <= (others => (others => '0'));
        arb_count_d <= (others => '0');
        state_d   <= IDLE;
    end if;
end process;

```

```

port_state_d <= 0;
ack_d      <= (others => '0');

elsif (clk = '1' and clk'event) then

    -- Delay all control signals by one cycle to compensate
    -- for write fifo delay.
    rnw_d      <= rnw_s;
    addr_d     <= addr_s;
    arb_count_d <= arb_count;
    state_d    <= state;
    port_state_d <= port_state;
    ack_d      <= ack_s;

    if (state_d = IDLE) then

        -- When idle deselect memory.
        mem_sel  <= '0';
        mem_rnw  <= '1';
        mem_addr <= (others => '0');
        mem_wdata <= (others => '0');
        mem_wport <= (others => '0');
        mem_wdaddr <= (others => '0');

    elsif (state_d = GRANT) then

        for i in 0 to NUM_PORTS-1 loop

            -- If this port is active service it.
            if (port_state_d = i) then

                -- When granted select memory.
                if (ack_d /= 0) then
                    mem_sel  <= '1';
                    mem_rnw  <= rnw_d(i);
                    mem_addr <= addr_d(i);
                end if;
                mem_wdata <= wdata_s(i);
                mem_wport <= conv_std_logic_vector(i, bit_width(NUM_PORTS));
                mem_wdaddr <= arb_count_d;

            end if;

        end loop;

    end if;

end process;

-- This is a sequential mux to routes out the read data to the proper port.
memory_input_proc : process(clk, reset)
begin
    if (reset = '1') then
        for i in 0 to NUM_PORTS-1 loop
            rdata_s(i) <= (others => '0');
            rval(i) <= '0';
            rwdaddr_s(i) <= (others => '0');
        end loop;
    elsif (clk = '1' and clk'event) then
        rval <= (others => '0');
        for i in 0 to NUM_PORTS-1 loop
            if (mem_rport = conv_std_logic_vector(i, bit_width(NUM_PORTS))) then
                rdata_s(i) <= mem_rdata;
            end if;
        end loop;
    end if;
end process;

```

```

        rval(i)      <= mem_dval;
        rwdaddr_s(i) <= mem_rwdaddr;
    end if;
end loop;
end if;
end process;

end rtl;

```

C.15 ZBT WIDTH CONVERSION

This VHDL file converts the internal data path's width to the physical memory's data path width.

```

-----
--
--
-- Filename      : zbt_width_conversion.vhd
--
-- Initial Date  : May 23 2007
--
-- Author        : James Ryan Warner
--
-- Description    : This block converts the incoming data vector to
--                  the appropriate memory vector width.
--
-----
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library work;
use work.zbt_ctrl_pkg.all;

entity zbt_width_conversion is
    generic (
        ADDR_WIDTH      : integer;
        DATA_WIDTH     : integer;
        MEMORY_WIDTH     : integer;
        BYTE_WIDTH       : integer;
        MAX_BURST_SIZE   : integer;
        MAX_MEM_BURST_SIZE : integer
    );
    port (
        -- Reset/Clock
        reset : in std_logic;      -- Async Reset.
        clk   : in std_logic;      -- Memory clock

```

```

-- Arbiter interface to port cmd fifo
arb_size_i      : in  std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto
0);
arb_addr_i      : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
arb_size_o      : out std_logic_vector(bit_width(MAX_MEM_BURST_SIZE)-1
downto 0);
arb_addr_o      : out std_logic_vector(ADDR_WIDTH-1 downto 0);

-- Arbiter interface to write fifo
arb_wdata_i     : in  std_logic_vector(DATA_WIDTH-1 downto 0);
arb_wdata_pop_i : in  std_logic;
arb_wdata_o     : out std_logic_vector(MEMORY_WIDTH-1 downto 0);
arb_wdata_pop_o : out std_logic;

-- Arbiter interface to port read fifo
arb_rdata_i     : in  std_logic_vector(MEMORY_WIDTH-1 downto 0);
arb_dval_i     : in  std_logic;
arb_rwdaddr_i   : in  std_logic_vector(bit_width(MAX_MEM_BURST_SIZE)-1
downto 0);
arb_rdata_o     : out std_logic_vector(DATA_WIDTH-1 downto 0);
arb_dval_o     : out std_logic;
arb_rwdaddr_o   : out std_logic_vector(bit_width(MAX_BURST_SIZE)-1 downto
0)
);

```

end zbt_width_conversion;

architecture rtl of zbt_width_conversion is

```

-- Various constants for vector widths and fifo widths.
constant NUM_BURST_BITS : integer := bit_width(MAX_BURST_SIZE);
constant DATA_RATIO    : integer := (DATA_WIDTH/MEMORY_WIDTH);
constant DATA_RATIO_BITS : integer := bit_width(DATA_RATIO);
constant ARB_SIZE_BITS  : integer := bit_width(MAX_MEM_BURST_SIZE);

signal write_count      : std_logic_vector(DATA_RATIO_BITS-1 downto 0);
signal write_count_d    : std_logic_vector(DATA_RATIO_BITS-1 downto 0);
signal read_count       : std_logic_vector(DATA_RATIO_BITS-1 downto 0);

begin

byte_address_shift_on : if (MEMORY_WIDTH > BYTE_WIDTH) generate
  arb_addr_o(ADDR_WIDTH-bit_width(MEMORY_WIDTH/BYTE_WIDTH)-1 downto 0)
<= arb_addr_i(ADDR_WIDTH-1 downto bit_width(MEMORY_WIDTH/BYTE_WIDTH));
  arb_addr_o(ADDR_WIDTH-1 downto ADDR_WIDTH-bit_width(MEMORY_WIDTH/BYTE_WIDTH))
<= (others => '0');
end generate;

byte_address_shift_off : if (MEMORY_WIDTH <= BYTE_WIDTH) generate
  arb_addr_o <= arb_addr_i;
end generate;

width_expand_on : if (DATA_WIDTH > MEMORY_WIDTH) generate

-- Wire out command signals.
arb_size_o(ARB_SIZE_BITS-1 downto NUM_BURST_BITS+DATA_RATIO_BITS) <=
(others => '0');
arb_size_o(NUM_BURST_BITS+DATA_RATIO_BITS-1 downto DATA_RATIO_BITS) <=
arb_size_i(NUM_BURST_BITS-1 downto 0);
arb_size_o(DATA_RATIO_BITS-1 downto 0) <=
(others => '1');

-- Simple counter to pop data from the fifo.

```

```

write_pop_prc : process (clk, reset)
begin
    -- Write counter.
    if (reset = '1') then
        write_count    <= (others => '0');
        write_count_d  <= (others => '0');
    elsif (clk = '1' and clk'event) then
        if (arb_wdata_pop_i = '1') then
            -- Increment write counter.
            write_count    <= write_count + 1;
            if (write_count = conv_std_logic_vector(DATA_RATIO-
1, bit_width(DATA_RATIO))) then
                write_count    <= (others => '0');
            end if;
        end if;
        write_count_d <= write_count;
    end if;
end process;

write_mux_prc : process (arb_wdata_pop_i, arb_wdata_i, write_count)
begin
    -- If this is the last word then pop the fifo.
    arb_wdata_pop_o <= '0';
    if (arb_wdata_pop_i = '1' and (write_count = 0)) then
        arb_wdata_pop_o <= '1';
    end if;

    -- Send out part of data word, this is simply a combinational mux.
    arb_wdata_o <= (others => '0');
    for i in 0 to DATA_RATIO-1 loop
        if (conv_integer(write_count_d) = i) then
            arb_wdata_o <= arb_wdata_i(((i+1) * MEMORY_WIDTH) - 1 downto (i *
MEMORY_WIDTH));
        end if;
    end loop;

end process;

-- Simple counter to pop data from the fifo.
read_push_prc : process (clk, reset)
begin
    if (reset = '1') then
        read_count    <= (others => '0');
        arb_dval_o    <= '0';
        arb_rdata_o    <= (others => '0');
        arb_rwdaddr_o <= (others => '0');
    elsif (clk = '1' and clk'event) then

        arb_dval_o    <= '0';

        -- Read counter
        if (arb_dval_i = '1') then

            -- Increment read counter.
            read_count <= read_count + 1;
            if (read_count = conv_std_logic_vector(DATA_RATIO-
1, bit_width(DATA_RATIO))) then
                -- If this is the last word then push the fifo.
                arb_dval_o <= '1';
                read_count <= (others => '0');
            end if;

            -- Send out part of data word.

```

```

        for i in 0 to DATA_RATIO-1 loop
            if (conv_integer(read_count) = i) then
                arb_rdata_o(((i+1) * MEMORY_WIDTH) - 1) downto (i *
MEMORY_WIDTH) <= arb_rdata_i;
                arb_rwdaddr_o <= arb_rwdaddr_i (NUM_BURST_BITS+DATA_RATIO_BITS- 1
downto DATA_RATIO_BITS);
            end if;
        end loop;

    end if;

end if;
end process;

end generate;

width_expand_off : if (DATA_WIDTH <= MEMORY_WIDTH) generate
width_not_max_on : if (ARB_SIZE_BITS > NUM_BURST_BITS) generate
    arb_size_o (ARB_SIZE_BITS-1 downto NUM_BURST_BITS) <= (others => '0');
end generate;
arb_size_o (NUM_BURST_BITS-1 downto 0) <= arb_size_i;
arb_wdata_o <= arb_wdata_i;
arb_wdata_pop_o <= arb_wdata_pop_i;
arb_rdata_o <= arb_rdata_i;
arb_dval_o <= arb_dval_i;
arb_rwdaddr_o <= arb_rwdaddr_i;
arb_rwdaddr_i (NUM_BURST_BITS-1 downto 0);

end generate;

end rtl;

```

C.16 ZBT MEMORY CONTROLLER PACKAGE

This VHDL file contains functions needed by the various blocks of the ZBT memory controller.

```

-----
--
--
-- Filename      : zbt_ctrl_pkg.vhd
--
-- Initial Date : May 23 2007
--
-- Author       : James Ryan Warner
--
-- Description  : Type, Constants and Functions needed for the zbt controller.
--
-----
--

library ieee;
use ieee.std_logic_1164.all;

```

```

package zbt_ctrl_pkg is

    constant MAX_NUM_INTF           : integer := 4;
    constant DEFAULT_ADDR_WIDTH    : integer := 20;
    constant DEFAULT_MEMORY_WIDTH  : integer := 18;
    constant DEFAULT_BYTE_WIDTH    : integer := 8;
    constant DEFAULT_DATA_DELAY    : integer := 2;

    type integer_array_t is array (0 to MAX_NUM_INTF-1) of integer;

    constant DEFAULT_DATA_WIDTH    : integer_array_t := (others =>
18);
    constant DEFAULT_MAX_BURST_SIZE : integer_array_t := (others =>
4);
    constant DEFAULT_CMD_FIFO_DEPTH : integer_array_t := (others =>
16);
    constant DEFAULT_WRITE_FIFO_DEPTH : integer_array_t := (others =>
64);
    constant DEFAULT_READ_FIFO_DEPTH : integer_array_t := (others =>
64);
    constant DEFAULT_CMD_FIFO_AFULL_THRESH : integer_array_t := (others =>
8);
    constant DEFAULT_WRITE_FIFO_AFULL_THRESH : integer_array_t := (others =>
32);
    constant DEFAULT_READ_FIFO_AFULL_THRESH : integer_array_t := (others =>
32);
    constant DEFAULT_FIFO_DUAL_CLOCK      : integer_array_t := (others =>
0);

    function bit_width (value : integer range 0 to 65535) return integer;

    function max_size (value : integer_array_t;
                      nports: integer) return integer;

    function total_size (value : integer_array_t;
                        nports: integer) return integer;

    function top_index (index : integer;
                       value : integer_array_t) return integer;

    function top_index (index : integer;
                       value : integer) return integer;

    function bottom_index (index : integer;
                          value : integer_array_t) return integer;

    function bottom_index (index : integer;
                          value : integer) return integer;

    function conv_array_bit_width ( value : integer_array_t;
                                   nports : integer ) return integer_array_t;

end zbt_ctrl_pkg;

package body zbt_ctrl_pkg is

    function bit_width (value : integer range 0 to 65535)
    return integer is
    begin
        for i in 0 to 65535 loop
            if (value <= 2**i) then
                if (i = 0) then return 1; end if;
                return i;
            end if;
        end loop;
    end function;

```



```

    end loop;
    return 65535;
end function bit_width;

function max_size (value : integer_array_t;
                  nports: integer)
    return integer is
        variable max : integer := 0;
begin
    for i in 0 to nports-1 loop
        if value(i) > max then max := value(i); end if;
    end loop;
    return max;
end function;

function total_size (value : integer_array_t;
                    nports: integer)
    return integer is
        variable total : integer := 0;
begin
    for i in 0 to nports-1 loop
        total := value(i) + total;
    end loop;
    return total;
end function;

function top_index (index : integer;
                   value : integer_array_t)
    return integer is
        variable total : integer := 0;
begin
    for i in 0 to index loop
        total := value(i) + total;
    end loop;
    return total - 1;
end function;

function top_index (index : integer;
                   value : integer)
    return integer is
        variable total : integer := 0;
begin
    for i in 0 to index loop
        total := value + total;
    end loop;
    return total - 1;
end function;

function bottom_index (index : integer;
                      value : integer_array_t)
    return integer is
        variable total : integer := 0;
begin
    total := top_index(index, value);
    return ((total+1) - value(index));
end function;

function bottom_index (index : integer;
                      value : integer)
    return integer is
        variable total : integer := 0;
begin
    total := top_index(index, value);
    return ((total+1) - value);
end function;

```

```

end function;

function conv_array_bit_width( value : integer_array_t;
                               nports: integer )
    return integer_array_t is
    variable temp : integer_array_t;
begin
    for i in 0 to value'length-1 loop
        if (i > nports-1) then
            temp(i) := 0;
        else
            temp(i) := bit_width(value(i));
        end if;
    end loop;
    return temp;
end function;
end zbt_ctrl_pkg;

```

C.17 DVI PHYSICAL INTERFACE

This VHDL file handles driving the control signals to the DVI controller. It is hard wired for VGA pass-through.

```

-----
-- Filename   : dvi_intf.vhd
--
-- Date       : September 20 2007
--
-- Author     : James Warner
--
-- Desc       : This logic handles driving control signals
--             to the dvi controller. Right now, it
--             only handles vga passthrough mode.
-----

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library work;
use work.gpu_pkg.all;

entity dvi_intf is
    port(
        -- Clock, reset and enable signals
        clk           : in  std_logic;
        reset_n       : in  std_logic;
        enable        : in  std_logic;

        -- VGA input signals.

```

```

    vga_vsync_n      : in  std_logic;
    vga_hsync_n      : in  std_logic;
    vga_red           : in  std_logic_vector(7 downto 0);
    vga_green         : in  std_logic_vector(7 downto 0);
    vga_blue          : in  std_logic_vector(7 downto 0);
    vga_valid         : in  std_logic;

    -- Display driver and fifo status
    dvi_hsync_n       : out std_logic;
    dvi_vsync_n       : out std_logic;
    dvi_data_en       : out std_logic;
    dvi_data1         : out std_logic_vector(11 downto 0);
    dvi_data2         : out std_logic_vector(11 downto 0)
);
end dvi_intf;

architecture hdl of dvi_intf is
    -- State signals.
    type dvi_state_t is (IDLE, ACTIVE);
    signal dvi_state : dvi_state_t;

begin
    -- Green lsb and blue get sent on first cycle,
    -- then red and green msb.
    sdr_process : process (clk, reset_n)
    begin
        if (reset_n = '0') then
            dvi_state    <= IDLE;
            dvi_data1    <= (others => '0');
            dvi_data2    <= (others => '0');
            dvi_hsync_n  <= '1';
            dvi_vsync_n  <= '1';
            dvi_data_en  <= '0';
        elsif (clk = '1' and clk'event) then
            case dvi_state is
            when IDLE =>
                if (enable = '1') then
                    dvi_state    <= ACTIVE;
                end if;
            when ACTIVE =>
                dvi_hsync_n <= vga_hsync_n;
                dvi_vsync_n <= vga_vsync_n;
                dvi_data_en <= vga_valid;
                dvi_data1   <= vga_green(3 downto 0) & vga_blue;
                dvi_data2   <= vga_red & vga_green(7 downto 4);
            end case;
        end if;
    end process;
end hdl;

```

C.18 VGA FRAME READER

This VHDL file handles pulling a frame from zbt memory when the VGA interface asks for a new frame of data.

```
-----
-- Filename   : vga_frame_reader.vhd
--
-- Date       : Oct. 23 2007
--
-- Author     : James Warner
--
-- Desc       : This is glue logic which interfaces
--              between the memory controller and the vga
--              controller. It handles driving out the
--              address
-----

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_unsigned.all;

library work;
  use work.gpu_pkg.all;

entity vga_frame_reader is
  port(

    -- Clock, reset and enable signals
    vga_clk       : in  std_logic;
    reset_n       : in  std_logic;

    -- Vga controller interface.
    vga_fifo_full : in  std_logic;
    vga_data_val  : out std_logic;
    vga_data      : out std_logic_vector(23 downto 0);
    vga_eof       : in  std_logic;

    -- ZBt memory interface.
    mem_req       : out std_logic;
    mem_full      : in  std_logic;
    mem_addr      : out std_logic_vector(18 downto 0);
    mem_rpop      : out std_logic;
    mem_rdata     : in  std_logic_vector(35 downto 0);
    mem_reempty   : in  std_logic;
    mem_rafull    : in  std_logic

  );
end vga_frame_reader;

architecture hdl of vga_frame_reader is

  type mem_read_req_state_t is (WAIT_FOR_EOF, WAIT_FOR_NFULL, WRITE_REQ);
  signal mem_read_req_state : mem_read_req_state_t;
```

```

type scan_line_double_state_t is
(WAIT_FOR_EOF, WAIT_EMPTY, PIX_ODD_LINE_ODD, PIX_EVEN_LINE_ODD, PIX_ODD_LINE_EVE
N, PIX_EVEN_LINE_EVEN);
signal scan_line_double_state : scan_line_double_state_t;

signal mem_addr_i      : std_logic_vector(16 downto 0);
signal bram_we         : std_logic;
signal bram_re         : std_logic;
signal bram_addr      : std_logic_vector(bit_width(320)-1 downto 0);
signal bram_wdata     : std_logic_vector(35 downto 0);
signal bram_rdata     : std_logic_vector(35 downto 0);
signal line_count     : std_logic_vector(bit_width(240)-1 downto 0);
signal mem_we_dly     : std_logic;
signal mem_rdata_dly  : std_logic_vector(35 downto 0);
signal mem_rafull_meta : std_logic;
signal mem_rafull_sync : std_logic;

component bram_1port_1clk is
generic (
MEM_WIDTH : integer;
MEM_SIZE  : integer
);
port (
clk      : in  std_logic;
we       : in  std_logic;
addr     : in  std_logic_vector(bit_width(MEM_SIZE)-1 downto 0);
wdata    : in  std_logic_vector(MEM_WIDTH-1 downto 0);
rdata    : out std_logic_vector(MEM_WIDTH-1 downto 0)
);
end component;

begin

-- Small block ram used to store a scan line of data.
scan_line_bram : bram_1port_1clk
generic map (
MEM_WIDTH => 36,
MEM_SIZE  => 320
)
port map (
clk  => vga_clk,
we   => bram_we,
addr => bram_addr,
wdata => bram_wdata,
rdata => bram_rdata
);

-- Read almost full sync register, syncs read almost full signal to vga_clk
domain.
rafull_sync_prc : process(vga_clk, reset_n)
begin
if (reset_n = '0') then
mem_rafull_meta <= '0';
mem_rafull_sync <= '0';
elsif (vga_clk = '1' and vga_clk'event) then
mem_rafull_meta <= mem_rafull;
mem_rafull_sync <= mem_rafull_meta;
end if;
end process;

-- Process to drive out vga control signals.
vga_data_out_prc : process(vga_clk, reset_n)
begin

```

```

if (reset_n = '0') then
    vga_data_val <= '0';
    vga_data     <= (others => '0');
    mem_we_dly  <= '0';
    mem_rdata_dly <= (others => '0');
elsif (vga_clk = '1' and vga_clk'event) then

    vga_data_val <= '0';
    mem_we_dly   <= '0';

    if (bram_re = '1') then
        vga_data_val <= '1';
        vga_data     <= bram_rdata(17 downto 12) & '0' & '0' &
            bram_rdata(11 downto 6)  & '0' & '0' &
            bram_rdata(5 downto 0)   & '0' & '0';
    elsif (bram_we = '1') then
        vga_data_val <= '1';
        vga_data     <= mem_rdata(17 downto 12) & '0' & '0' &
            mem_rdata(11 downto 6)  & '0' & '0' &
            mem_rdata(5 downto 0)   & '0' & '0';
    mem_rdata_dly <= mem_rdata;
    mem_we_dly    <= '1';
    elsif (mem_we_dly = '1') then
        vga_data_val <= '1';
        vga_data     <= mem_rdata_dly(17 downto 12) & '0' & '0' &
            mem_rdata_dly(11 downto 6) & '0' & '0' &
            mem_rdata_dly(5 downto 0)  & '0' & '0';
    end if;

end if;

end process;

-- Process to handle line double as well as pulling data from memory.
mem_line_double_prc : process(vga_clk, reset_n)
begin
    if (reset_n = '0') then

        mem_rpop      <= '0';
        bram_we       <= '0';
        bram_re       <= '0';
        bram_addr     <= (others => '0');
        line_count    <= (others => '0');

    elsif (vga_clk = '1' and vga_clk'event) then

        -- Defaults
        mem_rpop      <= '0';
        bram_we       <= '0';
        bram_re       <= '0';

        case scan_line_double_state is
            when WAIT_FOR_EOF =>

                if (vga_eof = '1') then
                    scan_line_double_state <= WAIT_NEMPTY;
                end if;

            when WAIT_NEMPTY =>

                if (mem_reempty = '0' and vga_fifo_full = '0') then
                    mem_rpop      <= '1';
                    bram_addr     <= (others => '0');
                end if;
            end case;
        end process;

```

```

    line_count          <= (others => '0');
    scan_line_double_state <= PIX_ODD_LINE_ODD;
end if;

when PIX_ODD_LINE_ODD =>

    if (vga_fifo_afull = '0') then
        bram_we          <= '1';
        scan_line_double_state <= PIX_EVEN_LINE_ODD;
    end if;

when PIX_EVEN_LINE_ODD =>

    if (vga_fifo_afull = '0') then
    if (bram_addr < 319) then
        if (mem_reempty = '0' and vga_fifo_afull = '0') then
            mem_rpop          <= '1';
            bram_addr          <= bram_addr + 1;
            scan_line_double_state <= PIX_ODD_LINE_ODD;
        end if;
    else
        bram_addr          <= (others => '0');
        scan_line_double_state <= PIX_ODD_LINE_EVEN;
    end if;
end if;

when PIX_ODD_LINE_EVEN =>

    if (vga_fifo_afull = '0') then
        bram_re          <= '1';
        scan_line_double_state <= PIX_EVEN_LINE_EVEN;
    end if;

when PIX_EVEN_LINE_EVEN =>

    if (vga_fifo_afull = '0') then
    if (bram_addr < 319) then
        bram_re          <= '1';
        bram_addr          <= bram_addr + 1;
        scan_line_double_state <= PIX_ODD_LINE_EVEN;
    else
        if (line_count < 239) then
            if (mem_reempty = '0' and vga_fifo_afull = '0') then
                mem_rpop          <= '1';
                bram_re          <= '1';
                bram_addr          <= (others => '0');
                line_count          <= line_count + 1;
                scan_line_double_state <= PIX_ODD_LINE_ODD;
            end if;
        else
            bram_re          <= '1';
            bram_addr          <= (others => '0');
            line_count          <= (others => '0');
            scan_line_double_state <= WAIT_FOR_EOF;
        end if;
    end if;
end if;

end case;

end if;
end process;
bram_wdata <= mem_rdata;

```

```

-- Process to drive out memory requests to the memory controller.
mem_read_req_prc : process(vga_clk, reset_n)
begin
  if (reset_n = '0') then

    -- Reset
    mem_req          <= '0';
    mem_addr_i       <= (others => '0');
    mem_read_req_state <= WAIT_FOR_EOF;

  elsif (vga_clk = '1' and vga_clk'event) then

    -- Defaults
    mem_req <= '0';

    -- Memory Request state machine.
    case mem_read_req_state is

      when WAIT_FOR_EOF =>

        -- Wait for end of frame until starting to drive
        -- out read requests to the memory controller.
        -- Check if the memory controller is full.
        if (vga_eof = '1') then
          if (mem_afull = '0' and mem_rafull_sync = '0') then
            mem_req          <= '1';
            mem_addr_i       <= (others => '0');
            mem_read_req_state <= WRITE_REQ;
          else
            mem_read_req_state <= WAIT_FOR_NFULL;
          end if;
        end if;

      when WAIT_FOR_NFULL =>

        -- An EOF has been seen but we need to wait until
        -- the memory controller is not busy.
        if (mem_afull = '0' and mem_rafull_sync = '0') then
          mem_req          <= '1';
          mem_addr_i       <= (others => '0');
          mem_read_req_state <= WRITE_REQ;
        end if;

      when WRITE_REQ =>

        -- Wait for the memory controller not to be full to send
        -- read requests. Also check the vga fifo to assure it isn't
        -- begining to overflow.
        if (mem_afull = '0' and mem_rafull_sync = '0') then
          if (mem_addr_i = (320 * 240)-1) then
            mem_addr_i       <= (others => '0');
            mem_read_req_state <= WAIT_FOR_EOF;
          else
            mem_req          <= '1';
            mem_addr_i <= mem_addr_i + 1;
          end if;
        end if;

      end case;

    end if;
  end process;

  mem_addr <= mem_addr_i & "00";

```



```
end hdl;
```

C.19 VGA SYNC GENERATOR

This VHDL file handles creating the vertical and horizontal sync pulses used by today's standard VGA monitors.

```
-----  
-- Filename   : vga_sync_generator.vhd  
--  
-- Date       : Februrary 20 2005  
--  
-- Author     : James Warner  
--  
-- Desc       : This block is a counter with comparitors  
--              used to generate horizontal or vertical  
--              sync pulses for a graphi cal di spl ay.  
-----  
  
library ieee;  
    use ieee.std_logic_1164.all;  
    use ieee.std_logic_arith.all;  
    use ieee.std_logic_unsigned.all;  
  
library work;  
    use work.gpu_pkg.all;  
  
entity vga_sync_generator is  
    generic(  
        -- Generics for horizontal timing  
        PULSE_LENGTH : integer := 96;  
        FRONT_PORCH  : integer := 16;  
        ACTIVE_VIDEO : integer := 640;  
        BACK_PORCH   : integer := 48  
    );  
  
    port(  
        -- Clock, reset and enable signals  
        clk          : in  std_logic;  
        reset_n     : in  std_logic;  
        clk_enable   : in  std_logic;  
  
        -- Output signals  
        sync_n      : out std_logic;  
        gate        : out std_logic;  
        blank       : out std_logic;  
        count       : out std_logic_vector(bit_width(PULSE_LENGTH +  
        FRONT_PORCH + ACTIVE_VIDEO + BACK_PORCH) - 1 downto 0)  
    );  
  
end vga_sync_generator;
```

architecture synth of vga_sync_generator is

```
-- Constants for horizontal sync timing
constant TOTAL_COUNT : integer := PULSE_LENGTH + FRONT_PORCH + ACTIVE_VIDEO
+ BACK_PORCH;
constant BLANK_MIN   : integer := PULSE_LENGTH + BACK_PORCH;
constant BLANK_MAX   : integer := TOTAL_COUNT - FRONT_PORCH;

-- Internal Counters for horizontal and vertical sync
signal count_i       : std_logic_vector(bit_width(TOTAL_COUNT)-1 downto 0);

begin

-- Main State Machine for VGA timing
sync_counter_prc : process (clk, reset_n)
begin

-- Active Low Reset
if (reset_n = '0') then

-- Reset Defaults
count_i <= (others=>'0');

-- rising edge of the clk.
elsif (clk'event and clk = '1') then
if (clk_enable = '1') then

-- Count ever rising clock.
if (count_i /= TOTAL_COUNT-1) then
-- Internal counter counts until its maximum value.
count_i <= count_i + 1;
else
-- Reset counter to zero when it passes its max value.
count_i <= (others=>'0');
end if;
end if;
end if;
end process;

-- Async. signals gate, sync, and blank.
gate  <= '1' when (count_i = TOTAL_COUNT-1) else '0';
blank <= '1' when ((count_i < BLANK_MIN) or (count_i >= BLANK_MAX)) else
'0';
sync_n <= '0' when (count_i < PULSE_LENGTH) else '1';
count <= count_i;

end synth;
```

C.20 VGA CONTROLLER

This VHDL file drives the red, green, blue, vertical sync and horizontal sync to the VGA interface.

```

-- Filename   : vga_ctrl.vhd
--
-- Date      : February 20 2005
--
-- Author    : James Warner
--
-- Desc      : This logic drives the display VSYNC and HSYNC
--             signals. It also passes through the 3 bit,
--             red, green and blue signals.
-----

```

```

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_unsigned.all;

```

```

library work;
  use work.gpu_pkg.all;

```

```

entity vga_ctrl is
  generic(
    -- Number of bits per color
    NUM_COLOR_BITS : natural := 8;

    -- Video timing Generics.
    H_ACTIVE_VIDEO : integer := 640;
    H_PULSE_LENGTH : integer := 96;
    H_FRONT_PORCH  : integer := 16;
    H_BACK_PORCH   : integer := 48;
    V_ACTIVE_VIDEO : integer := 480;
    V_PULSE_LENGTH : integer := 2;
    V_FRONT_PORCH  : integer := 11;
    V_BACK_PORCH   : integer := 31;

    -- Pixel Fifo Generics
    FIFO_DEPTH      : integer := 16;
    FIFO_AFULL_THRESH : integer := 9;
    FIFO_AEMPTY_THRESH : integer := 8
  );

```

```

port(
  -- Clock, reset and enable signals
  vga_clk      : in  std_logic;
  reset_n     : in  std_logic;
  gpu_enable   : in  std_logic;

  -- Input pixel data.
  pixel_data_in : in  std_logic_vector((NUM_COLOR_BITS * 3) - 1 downto 0);
  pixel_wr_req  : in  std_logic;

  -- Signals to the display
  vsync_n      : out std_logic;
  hsync_n      : out std_logic;
  red_value    : out std_logic_vector(NUM_COLOR_BITS-1 downto 0);
  green_value  : out std_logic_vector(NUM_COLOR_BITS-1 downto 0);
  blue_value   : out std_logic_vector(NUM_COLOR_BITS-1 downto 0);
  vga_valid    : out std_logic;

  -- Display driver and fifo status
  pixel_fifo_full : out std_logic;
  pixel_fifo_empty : out std_logic;

```

```

        pixel_fifo_full : out std_logic;
        pixel_fifo_empty: out std_logic;
        pixel_eof       : out std_logic
    );
end vga_ctrl;

architecture hdl of vga_ctrl is
    -- Constants
    constant PIXEL_SIZE : integer := NUM_COLOR_BITS * 3;
    constant FIFO_BITS  : integer := bit_width(FIFO_DEPTH);
    constant H_TOTAL    : integer := H_ACTIVE_VIDEO + H_PULSE_LENGTH +
H_FRONT_PORCH + H_BACK_PORCH;
    constant V_TOTAL    : integer := V_ACTIVE_VIDEO + V_PULSE_LENGTH +
V_FRONT_PORCH + V_BACK_PORCH;

    -- State Type and State Vector
    type gpu_drive_state_t is (GPU_IDLE, GPU_PRE_DRIVE, GPU_DRIVE);
    signal gpu_drive_state : gpu_drive_state_t;

    -- Internal Counters for horizontal and vertical sync
    signal hcount          : std_logic_vector(bit_width(H_TOTAL)-1 downto
0);
    signal vcount          : std_logic_vector(bit_width(V_TOTAL)-1 downto
0);

    -- Horizontal and Vertical generator enable signals.
    signal hsync_enable   : std_logic;
    signal vsync_enable   : std_logic;
    signal eof_enable     : std_logic;

    -- Blanking signals
    signal hblank         : std_logic;
    signal vblank         : std_logic;
    signal reset          : std_logic;

    -- Pixel Fifo internal signals
    signal pixel_read_data : std_logic_vector((NUM_COLOR_BITS*3)-1 downto
0);
    signal pixel_read_req  : std_logic;

    -- Pixel Fifo internal signals
    signal pixel_fifo_empty_s : std_logic;
    signal pixel_fifo_aempty_s : std_logic;
    signal pixel_fifo_full_s  : std_logic;
    signal pixel_fifo_full_s  : std_logic;

    -- Output register
    signal vsync_n_i        : std_logic;
    signal hsync_n_i        : std_logic;
    signal red_value_i      : std_logic_vector(NUM_COLOR_BITS-1 downto 0);
    signal green_value_i    : std_logic_vector(NUM_COLOR_BITS-1 downto 0);
    signal blue_value_i     : std_logic_vector(NUM_COLOR_BITS-1 downto 0);
    signal vga_val_id_i     : std_logic;

    -- Sync pulse generator component.
    component vga_sync_generator
        generic (
            -- Generics for horizontal timing
            PULSE_LENGTH : integer;
            FRONT_PORCH  : integer;
            ACTIVE_VIDEO  : integer;

```

```

    BACK_PORCH : integer
);

port(
-- Clock, reset and enable signals
  clk       : in  std_logic;
  reset_n   : in  std_logic;
  clk_enable : in  std_logic;

  -- Output signals
  sync_n    : out std_logic;
  gate      : out std_logic;
  blank     : out std_logic;
  count     : out std_logic_vector(bit_width(PULSE_LENGTH +
FRONT_PORCH + ACTIVE_VIDEO + BACK_PORCH) - 1 downto 0)
);
end component;

-- Pixel Fifo component.
component fifo_1clk
generic (
  FIFO_WIDTH      : integer; -- Fifo with in bits
  FIFO_DEPTH      : integer; -- Fifo depth in FIFO_WIDTH sized words.
  FIFO_AFULL_THRESH : integer; -- Almost empty level threshold
  FIFO_AEMPTY_THRESH : integer; -- Almost empty level threshold
  FIFO_FALL_THROUGH : integer
);

port(
-- Clock and reset
  reset : in  std_logic;
  clk   : in  std_logic;

  -- Control signals
  push  : in  std_logic;
  pop   : in  std_logic;

  -- Read write data
  wdata : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
  rdata : out std_logic_vector(FIFO_WIDTH-1 downto 0);

  -- Status flags.
  afull : out std_logic;
  aempty : out std_logic;
  empty : out std_logic;
  full  : out std_logic
);
end component;

begin

-- Instantiate Horizontal Sync Generator
hsync_gen_0 : vga_sync_generator
generic map (
  PULSE_LENGTH => H_PULSE_LENGTH,
  FRONT_PORCH  => H_FRONT_PORCH,
  ACTIVE_VIDEO => H_ACTIVE_VIDEO,
  BACK_PORCH   => H_BACK_PORCH
)
port map (
  clk       => vga_clk,
  reset_n   => reset_n,
  clk_enable => hsync_enable,
  sync_n    => hsync_n_i,

```

```

        gate      => vsync_enable,
        blank     => hblank,
        count     => hcount
    );

-- Instantiate Vertical Sync Generator
vsync_gen_0 : vga_sync_generator
    generic map (
        PULSE_LENGTH => V_PULSE_LENGTH,
        FRONT_PORCH  => V_FRONT_PORCH,
        ACTIVE_VIDEO => V_ACTIVE_VIDEO,
        BACK_PORCH   => V_BACK_PORCH
    )
    port map (
        clk      => vga_clk,
        reset_n  => reset_n,
        clk_enable => vsync_enable,
        sync_n   => vsync_n_i,
        gate     => eof_enable,
        blank    => vblank,
        count    => vcount
    );

-- Instanciate Pixel Fifo
reset <= not reset_n;
pixel_fifo_0 : fifo_1clk
    generic map (
        FIFO_WIDTH      => PIXEL_SIZE,
        FIFO_DEPTH      => FIFO_DEPTH,
        FIFO_AFULL_THRESH => FIFO_AFULL_THRESH,
        FIFO_AEMPTY_THRESH => FIFO_AEMPTY_THRESH,
        FIFO_FALL_THROUGH => 1
    )
    port map (
        reset => reset,
        clk   => vga_clk,
        push  => pixel_wr_req,
        pop   => pixel_read_req,
        wdata => pixel_data_in,
        rdata => pixel_read_data,
        afull => pixel_fifo_afull_s,
        aempty => pixel_fifo_aempty_s,
        empty => pixel_fifo_empty_s,
        full  => pixel_fifo_full_s
    );

-- End of frame or reset fifo signal.
pixel_read_req <= not vblank and not hblank;
pixel_fifo_empty <= pixel_fifo_empty_s;
pixel_fifo_full <= pixel_fifo_full_s;
pixel_fifo_afull <= pixel_fifo_afull_s;
pixel_fifo_aempty <= pixel_fifo_aempty_s;
pixel_eof <= eof_enable and vsync_enable;

-- Main State Machine for VGA timing
gpu_drive_state_prc : process (vga_clk, reset_n)
begin
    -- Active Low Reset
    if (reset_n = '0') then
        -- Reset Defaults
        gpu_drive_state <= GPU_IDLE;
    end if;
end process;

```

```

hsync_enable      <= '0';
red_value_i      <= (others=>'0');
blue_value_i     <= (others=>'0');
green_value_i    <= (others=>'0');
vga_valid_i      <= '0';

-- States register on rising edge of the clk.
elsif (vga_clk'event and vga_clk = '1') then

  case gpu_drive_state is

    when GPU_IDLE =>

      -- Wait for enable signal.
      -- Once enable is set start loading in pixel data.
      if (gpu_enable = '1') then
        gpu_drive_state <= GPU_PRE_DRIVE;
      end if;

    when GPU_PRE_DRIVE =>

      -- Assert hsync_enable which will kick off the
      -- GPU Driver.
      hsync_enable <= '1';
      if (eof_enable = '1' and vsync_enable = '1') then
        gpu_drive_state <= GPU_DRIVE;
      end if;

    when GPU_DRIVE =>

      if (hblank = '1' or vblank = '1') then
        -- Blank Pixels if outside active video area.
        red_value_i   <= (others=>'0');
        green_value_i <= (others=>'0');
        blue_value_i  <= (others=>'0');
        vga_valid_i   <= '0';
      else
        -- Drive out pixels if in active screen area.
        red_value_i   <= pixel_read_data(NUM_COLOR_BITS*1-1 downto 0);
        green_value_i <= pixel_read_data(NUM_COLOR_BITS*2-1 downto
NUM_COLOR_BITS*1);
        blue_value_i  <= pixel_read_data(NUM_COLOR_BITS*3-1 downto
NUM_COLOR_BITS*2);
        vga_valid_i   <= '1';
      end if;

    end case;
  end if;
end process;

reg_output_prc : process (vga_clk, reset_n)
begin

  if (reset_n = '0') then

    -- Reset Output Registers
    vsync_n      <= '1';
    hsync_n      <= '1';
    red_value    <= (others=>'0');
    green_value  <= (others=>'0');
    blue_value   <= (others=>'0');
    vga_valid    <= '0';

  elsif (vga_clk'event and vga_clk='1') then

```

```

vsync_n    <= vsync_n_i;
hsync_n    <= hsync_n_i;
red_value  <= red_value_i;
green_value <= green_value_i;
blue_value <= blue_value_i;
vga_valid  <= vga_valid_i;

end if;

end process;

end hdl;

```

C.21 GRAPHICS PIPELINE TESTBENCH

This VHDL is used to test the actual graphics pipeline by loading objects into the graphics pipeline and using a PPM file generator to view the results.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library work;
use work.zbt_ctrl_pkg.all;

entity video_control_top_tb is
end video_control_top_tb;

architecture test of video_control_top_tb is

-- Constants
constant sysCLK    : time := 5 ns; -- 100 Mhz
constant zbtCLK    : time := 5 ns; -- 100 Mhz
constant vgaCLK    : time := 20 ns; -- 25 Mhz
constant dviCLK    : time := 10 ns; -- 50 Mhz
constant tsetup    : time := 1 ns;
constant boardDelay : time := 1 ns;

-- Tieoffs in case you need them...
signal vdd          : std_logic;
signal gnd          : std_logic_vector(63 downto 0);

component ppm_gen is
generic(
    X_DIM : integer := 640;
    Y_DIM : integer := 480;
    PATH  : string := "./ppm_frames/output_frames/"
);
port (
    pixel_clk : in std_logic;

```



```

        enable      : in std_logic;
        red         : in std_logic_vector(7 downto 0);
        green       : in std_logic_vector(7 downto 0);
        blue        : in std_logic_vector(7 downto 0);
        push        : in std_logic
    );
end component;

component cy7c1352
generic (
    -- Constant parameters
    addr_bits : INTEGER := 18;
    data_bits : INTEGER := 36;

    -- Timing parameters for -5 (133 Mhz)
    tCYC      : TIME := 7.5 ns;
    tCH       : TIME := 3.0 ns;
    tCL       : TIME := 3.0 ns;
    tCO       : TIME := 4.0 ns;
    tAS       : TIME := 1.5 ns;
    tCENS     : TIME := 1.5 ns;
    tWES      : TIME := 1.5 ns;
    tDS       : TIME := 1.5 ns;
    tAH       : TIME := 0.5 ns;
    tCENH     : TIME := 0.5 ns;
    tWEH      : TIME := 0.5 ns;
    tDH       : TIME := 0.5 ns
);

-- Port Declarations
port (
I/O   Dq          : INOUT STD_LOGIC_VECTOR ((data_bits - 1) DOWNTO 0); -- Data

    Addr      : IN STD_LOGIC_VECTOR ((addr_bits - 1) DOWNTO 0); -- Address
    Mode      : IN STD_LOGIC           := '1';                    -- Burst Mode
    Clk       : IN STD_LOGIC;                    -- Clk
    CEN_n     : IN STD_LOGIC;                    -- CEN#
    AdvLd_n  : IN STD_LOGIC;                    -- Adv/Ld#
    Bwa_n    : IN STD_LOGIC;                    -- Bwa#
    Bwb_n    : IN STD_LOGIC;                    -- Bwb#
    Rw_n     : IN STD_LOGIC;                    -- RW#
    Oe_n     : IN STD_LOGIC;                    -- OE#
    Ce1_n    : IN STD_LOGIC;                    -- CE1#
    Ce2      : IN STD_LOGIC;                    -- CE2
    Ce3_n    : IN STD_LOGIC;                    -- CE3#
    Zz       : IN STD_LOGIC                    -- Snooze Mode
);
end component;

component video_control_top is
port(

    -- Reset/Clock
    reset      : in std_logic; -- Async Reset.
    sys_clk    : in std_logic; -- System clock.
    zbt_clk    : in std_logic; -- ZBT memory clock.
    vga_clk    : in std_logic; -- Vga clock.

    -- VGA enable signals.
    vga_enable : in std_logic;

    -- New frame trigger signals.
    gpu_enable : in std_logic;

```

```

-- Background
background : in std_logic_vector(35 downto 0);

-- End of file
eof        : out std_logic;

-- Matrix input
matrix_we  : in std_logic;
matrix_sel : in std_logic_vector(1 downto 0);
matrix_waddr : in std_logic_vector(1 downto 0);
matrix_wdata : in std_logic_vector(127 downto 0);

-- Pixel Pipe.
x_in      : in std_logic_vector(31 downto 0);
y_in      : in std_logic_vector(31 downto 0);
z_in      : in std_logic_vector(31 downto 0);
w_in      : in std_logic_vector(31 downto 0);
color     : in std_logic_vector(17 downto 0);
pix_valid : in std_logic;
pix_ready : out std_logic;

-- Clipping Maximum
zmax      : in std_logic_vector(31 downto 0);

-- CPU Interface port.
cpu_sel   : in std_logic;
cpu_we    : in std_logic;
cpu_addr  : in std_logic_vector(19 downto 0);
cpu_wdata : in std_logic_vector(35 downto 0);
cpu_wdone : out std_logic;
cpu_dval  : out std_logic;
cpu_rdata : out std_logic_vector(35 downto 0);

-- ZBT interface
zbt_cen   : out std_logic;
zbt_wen   : out std_logic;
zbt_oen   : out std_logic;
zbt_ts    : out std_logic;
zbt_wdata : out std_logic_vector(35 downto 0);
zbt_addr  : out std_logic_vector(17 downto 0);
zbt_rdata : in std_logic_vector(35 downto 0);

-- Dvi interface signals.
dvi_hsync_n : out std_logic;
dvi_vsync_n : out std_logic;
dvi_data_en : out std_logic;
dvi_data1   : out std_logic_vector(11 downto 0);
dvi_data2   : out std_logic_vector(11 downto 0)
);
end component;

component user_logic is
port(

-- Video output enable.
video_enable_o      : out std_logic;
gpu_enable_o        : out std_logic;

-- Background color
background_o        : out std_logic_vector(35 downto 0);

-- End of file
eof_i               : in std_logic;

```

```

-- Matrix Input
matrix_we           : out std_logic;
matrix_sel          : out std_logic_vector(1 downto 0);
matrix_waddr        : out std_logic_vector(1 downto 0);
matrix_wdata        : out std_logic_vector(127 downto 0);

-- Pixel input
x_in_o              : out std_logic_vector(31 downto 0);
y_in_o              : out std_logic_vector(31 downto 0);
z_in_o              : out std_logic_vector(31 downto 0);
w_in_o              : out std_logic_vector(31 downto 0);
color_in_o          : out std_logic_vector(17 downto 0);
point_trig_o        : out std_logic;

-- ZBT DCM lock
zbt_dcm_lock        : in  std_logic;

-- Clipping Zmax
zmax_o              : out std_logic_vector(31 downto 0);

-- Debug zbt memory interface.
zbt_mbox_sel        : out std_logic;
zbt_mbox_we         : out std_logic;
zbt_mbox_addr       : out std_logic_vector(19 downto 0);
zbt_mbox_wdata      : out std_logic_vector(35 downto 0);
zbt_mbox_rdata      : in  std_logic_vector(35 downto 0);
zbt_mbox_wdone      : in  std_logic;
zbt_mbox_dval       : in  std_logic;

-- OPB interface.
Bus2IP_Clk          : in  std_logic;
Bus2IP_Reset        : in  std_logic;
Bus2IP_Addr         : in  std_logic_vector(0 to 31);
Bus2IP_CS           : in  std_logic_vector(0 to 0);
Bus2IP_RNW          : in  std_logic;
Bus2IP_Data         : in  std_logic_vector(0 to 31);
Bus2IP_BE           : in  std_logic_vector(0 to 3);
IP2Bus_Data         : out std_logic_vector(0 to 31);
IP2Bus_RdAck        : out std_logic;
IP2Bus_WrAck        : out std_logic;
IP2Bus_Error        : out std_logic;
IP2Bus_IntrEvent    : out std_logic_vector(0 to 0)
);

end component;

signal vga_enable    : std_logic;
signal gpu_enable    : std_logic;
signal background    : std_logic_vector(35 downto 0);
signal eof           : std_logic;

signal zbt_adv       : std_logic;
signal zbt_addr      : std_logic_vector(17 downto 0);
signal zbt_bwn       : std_logic_vector( 1 downto 0);
signal zbt_wen       : std_logic;
signal zbt_oen       : std_logic;
signal zbt_cen_i     : std_logic;
signal zbt_ts        : std_logic;
signal zbt_zz        : std_logic;
signal zbt_mode      : std_logic;
signal zbt_cen       : std_logic;
signal zbt_ce        : std_logic_vector(2 downto 0);
signal zbt_dq        : std_logic_vector(35 downto 0);

```

```

signal zbt_wdata      : std_logic_vector(35 downto 0);
signal zbt_rdata      : std_logic_vector(35 downto 0);

signal reset          : std_logic := '1';
signal sys_clk        : std_logic := '0';
signal zbt_clk        : std_logic := '0';
signal vga_clk        : std_logic := '0';

signal cpu_sel        : std_logic;
signal cpu_we         : std_logic;
signal cpu_addr       : std_logic_vector(19 downto 0);
signal cpu_wdata      : std_logic_vector(35 downto 0);
signal cpu_wdone      : std_logic;
signal cpu_dval       : std_logic;
signal cpu_rdata      : std_logic_vector(35 downto 0);

signal dvi_clk        : std_logic := '1';
signal dvi_hsync_n    : std_logic;
signal dvi_vsync_n    : std_logic;
signal dvi_data_en    : std_logic;
signal dvi_data1      : std_logic_vector(11 downto 0);
signal dvi_data2      : std_logic_vector(11 downto 0);

signal ppm_red        : std_logic_vector(7 downto 0);
signal ppm_green      : std_logic_vector(7 downto 0);
signal ppm_blue       : std_logic_vector(7 downto 0);
signal ppm_push       : std_logic;
type ppm_state_t is (FIRST, SECOND);
signal ppm_state      : ppm_state_t;

signal ip_addr        : std_logic_vector(31 downto 0);
signal ip_rnw         : std_logic;
signal ip_rack        : std_logic;
signal ip_wack        : std_logic;
signal ip_wdata       : std_logic_vector(31 downto 0);
signal ip_be          : std_logic_vector(3 downto 0);
signal ip_cs          : std_logic_vector(0 downto 0);
signal ip_rdata       : std_logic_vector(31 downto 0);

signal x_in           : std_logic_vector(31 downto 0);
signal y_in           : std_logic_vector(31 downto 0);
signal z_in           : std_logic_vector(31 downto 0);
signal w_in           : std_logic_vector(31 downto 0);
signal color_in       : std_logic_vector(17 downto 0);
signal point_trig     : std_logic;

signal zmax           : std_logic_vector(31 downto 0);

signal matrix_we      : std_logic;
signal matrix_sel     : std_logic_vector(1 downto 0);
signal matrix_waddr   : std_logic_vector(1 downto 0);
signal matrix_wdata   : std_logic_vector(127 downto 0);

```

begin

```

-- Power supplies...
vdd    <= '1';
gnd    <= (others => '0');

-- Clock and reset generation.
sys_clk <= (not sys_clk) after sysCLK;
vga_clk <= (not vga_clk) after vgaCLK;
dvi_clk <= (not dvi_clk) after dviCLK;
zbt_clk <= (not zbt_clk) after zbtCLK;

```

```

reset    <= '1', '0' after (sysCLK * 25);

process(dvi_clk, reset)
begin
  if (reset = '1') then

    ppm_red    <= (others => '0');
    ppm_green  <= (others => '0');
    ppm_blue   <= (others => '0');
    ppm_push   <= '0';
    ppm_state  <= FIRST;

  elsif (dvi_clk = '1' and dvi_clk'event) then

    ppm_push <= '0';
    if (dvi_data_en = '1') then
      if (ppm_state = FIRST) then
        ppm_state      <= SECOND;
        ppm_red        <= dvi_data1(7 downto 0);
        ppm_green(3 downto 0) <= dvi_data1(11 downto 8);
      else
        ppm_state      <= FIRST;
        ppm_green(7 downto 4) <= dvi_data2(3 downto 0);
        ppm_blue       <= dvi_data2(11 downto 4);
        ppm_push       <= '1';
      end if;
    end if;

  end if;
end process;

u1_generator : ppm_gen
generic map(
  X_DIM => 640,
  Y_DIM => 480
)
port map(
  pixel_clk => dvi_clk,
  enable    => vga_enable,
  red       => ppm_red,
  green     => ppm_green,
  blue     => ppm_blue,
  push     => ppm_push
);

video_control_top_0 : video_control_top
port map(
  -- Reset/Clock
  reset    => reset,
  sys_clk  => sys_clk,
  zbt_clk  => zbt_clk,
  vga_clk  => vga_clk,

  -- VGA enable signals.
  vga_enable => vga_enable,

  -- New frame trigger signals.
  gpu_enable => gpu_enable,

  -- Background
  background => background,

```

```

-- End of frame.
eof      => eof,

-- Matrix input
matrix_we      => matrix_we,
matrix_sel     => matrix_sel,
matrix_waddr   => matrix_waddr,
matrix_wdata   => matrix_wdata,

-- Pixel Pipe.
x_in          => x_in,
y_in          => y_in,
z_in          => z_in,
w_in          => w_in,
color         => color_in,
pix_valid     => point_trig,
pix_ready     => open,

-- Clipping Zmax
zmax          => zmax,

-- CPU Interface port.
cpu_sel       => cpu_sel,
cpu_we        => cpu_we,
cpu_addr      => cpu_addr,
cpu_wdata     => cpu_wdata,
cpu_wdone     => cpu_wdone,
cpu_dval      => cpu_dval,
cpu_rdata     => cpu_rdata,

-- ZBT interface
zbt_cen       => zbt_cen_i,
zbt_wen       => zbt_wen,
zbt_oen       => zbt_oen,
zbt_ts        => zbt_ts,
zbt_wdata     => zbt_wdata,
zbt_addr      => zbt_addr,
zbt_rdata     => zbt_rdata,

-- DVI interface
dvi_hsync_n   => dvi_hsync_n,
dvi_vsync_n   => dvi_vsync_n,
dvi_data_en   => dvi_data_en,
dvi_data1     => dvi_data1,
dvi_data2     => dvi_data2

);

zbt_model_0 : cy7c1352
generic map(
  -- Constant parameters
  addr_bits => 18,
  data_bits => 36,

  -- Timing parameters for -5 (133 Mhz)
  tCYC      => 7.5 ns,
  tCH       => 3.0 ns,
  tCL       => 3.0 ns,
  tCO       => 4.0 ns,
  tAS       => 1.5 ns,
  tCENS     => 1.5 ns,
  tWES     => 1.5 ns,
  tDS       => 1.5 ns,

```

```

    tAH      => 0.5 ns,
    tCENH   => 0.5 ns,
    tWEH    => 0.5 ns,
    tDH     => 0.5 ns
)
-- Port Declarations
port map(
    Dq      => zbt_dq,
    Addr    => zbt_addr,
    Mode    => zbt_mode,
    Clk     => zbt_clk,
    CEN_n   => zbt_cen,      -- Clock enable...
    AdvLd_n => zbt_adv,
    Bwa_n   => zbt_bwn(0),
    Bwb_n   => zbt_bwn(1),
    Rw_n    => zbt_wen,
    Oe_n    => zbt_oen,
    Ce1_n   => zbt_ce(0),
    Ce2     => zbt_ce(1),
    Ce3_n   => zbt_ce(2),
    Zz      => zbt_zz
);

-- Tristate buffer for ZBT data...
zbt_dq    <= zbt_wdata when (zbt_ts = '0') else (others => 'Z');
zbt_rdata <= zbt_dq(35 downto 0);

zbt_mode  <= '0';
zbt_zz    <= '0';
zbt_ce(0) <= '0';
zbt_ce(1) <= '1';
zbt_ce(2) <= zbt_cen_i;
zbt_bwn   <= (others => '0');
zbt_adv   <= '0';
zbt_cen   <= '0';

u_regs_if0 : user_logic
    port map(
        -- Video output enable.
        video_enable_o => vga_enable,
        gpu_enable_o   => gpu_enable,

        -- Background color
        background_o   => background,

        -- End of frame
        eof_i          => eof,

        -- Matrix input
        matrix_we      => matrix_we,
        matrix_sel     => matrix_sel,
        matrix_waddr   => matrix_waddr,
        matrix_wdata   => matrix_wdata,

        -- Vector input
        x_in_o        => x_in,
        y_in_o        => y_in,
        z_in_o        => z_in,
        w_in_o        => w_in,
        color_in_o    => color_in,
        point_trig_o  => point_trig,

        -- Zmax Clipping

```

```

zmax_o          => zmax,

-- ZBT DCM lock
zbt_dcm_lock    => '1',

-- Debug zbt memory interface.
zbt_mbox_sel    => cpu_sel,
zbt_mbox_we     => cpu_we,
zbt_mbox_addr   => cpu_addr,
zbt_mbox_wdata  => cpu_wdata,
zbt_mbox_rdata  => cpu_rdata,
zbt_mbox_wdone  => cpu_wdone,
zbt_mbox_dval   => cpu_dval,

-- PLB interface.
Bus2IP_Clk      => sys_clk,
Bus2IP_Reset    => reset,
Bus2IP_Addr     => ip_addr,
Bus2IP_CS       => ip_cs,
Bus2IP_RNW      => ip_rnw,
Bus2IP_Data     => ip_wdata,
Bus2IP_BE       => ip_be,
IP2Bus_Data     => ip_rdata,
IP2Bus_RdAck    => ip_rack,
IP2Bus_WrAck    => ip_wack,
IP2Bus_Error    => open,
IP2Bus_IntrEvent => open

);

cpu_mem_write_prc : process

variable rslt_data : std_logic_vector(31 downto 0) := (others => '0');

procedure cpu_write_addr ( addr : in std_logic_vector(7 downto 0);
                          wdata : in std_logic_vector(31 downto 0)
                        ) is
begin
    wait until sys_clk = '0';
    ip_addr <= x"0000" & "00" & addr & "00";
    ip_be   <= (others => '1');
    ip_cs   <= "1";
    ip_rnw  <= '0';
    ip_wdata <= wdata;
    wait until (sys_clk = '0');
    ip_addr <= (others => '0');
    ip_be   <= (others => '0');
    ip_cs   <= "0";
    ip_rnw  <= '1';
    ip_wdata <= (others => '0');

end procedure;

procedure cpu_read_addr ( addr : in std_logic_vector(7 downto 0);
                        rdata : out std_logic_vector(31 downto 0)
                      ) is
begin
    wait until sys_clk = '0';
    ip_addr <= x"0000" & "00" & addr & "00";
    ip_be   <= (others => '1');
    ip_cs   <= "1";

```



```

ip_rnw    <= '1';
wait until (sys_clk = '0');
ip_addr  <= (others => '0');
ip_be    <= (others => '0');
ip_cs    <= "0";
ip_rnw   <= '1';
rdata    := ip_rdata;

end procedure;

begin

-- Initial conditions

-- Wait for reset
wait until reset = '0';
wait until sys_clk = '0';

-- Read Id.
cpu_read_addr(x"00", rslt_data);

-- Write and read scratch register.
cpu_write_addr(x"01", x"1234abcd");
cpu_read_addr(x"01", rslt_data);

-- Write in a screen.
for y in 0 to 239 loop
  for x in 0 to 319 loop
    cpu_write_addr(x"12", x"00000000");
    cpu_write_addr(x"13", x"00000000");
    cpu_write_addr(x"10", conv_std_logic_vector((x+(y*320))*4, 32));
  end loop;
end loop;

-- Write background register
cpu_write_addr(x"04", x"0000003F");
cpu_write_addr(x"05", x"00000000");

-- Zmax register
cpu_write_addr(x"D0", x"BDCCCCC");

-- Setup Default World translation to view coordinates matrix values.
cpu_write_addr(x"E0", x"3F800000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"00000000");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"3F800000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"00000001");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"3F800000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"00000002");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"3F800000");

```

```

cpu_write_addr(x"E4", x"00000003");

-- Setup Projection translation to screen coordinates matrix values.
cpu_write_addr(x"E0", x"431F8000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"431f8000");
cpu_write_addr(x"E4", x"0000000C");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"42EF0000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"42EF0000");
cpu_write_addr(x"E4", x"0000000D");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"3F800000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"0000000E");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"3F800000");
cpu_write_addr(x"E4", x"0000000F");

-- Enable GPU engine.
cpu_write_addr(x"03", x"00000002");

-- Enable VGA engine.
cpu_write_addr(x"03", x"00000003");

-- Draw a triangle
while (true) loop

    -- Wait for end of frame.
    wait until eof = '1';

    -- Setup Default View translation to screen coordinates matrix
    val ues.
cpu_write_addr(x"E0", x"3F800000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"00000004");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"3F800000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"00000005");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"3F800000");
cpu_write_addr(x"E3", x"00000000");
cpu_write_addr(x"E4", x"00000006");

cpu_write_addr(x"E0", x"00000000");
cpu_write_addr(x"E1", x"00000000");
cpu_write_addr(x"E2", x"00000000");
cpu_write_addr(x"E3", x"3F800000");
cpu_write_addr(x"E4", x"00000007");

```

values. -- Setup Default View translation to screen coordinates matrix

```
cpu_write_addr(x"E0", x"3F800000");  
cpu_write_addr(x"E1", x"00000000");  
cpu_write_addr(x"E2", x"00000000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"00000008");
```

```
cpu_write_addr(x"E0", x"00000000");  
cpu_write_addr(x"E1", x"3F800000");  
cpu_write_addr(x"E2", x"00000000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"00000009");
```

```
cpu_write_addr(x"E0", x"00000000");  
cpu_write_addr(x"E1", x"00000000");  
cpu_write_addr(x"E2", x"3F800000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"0000000A");
```

```
cpu_write_addr(x"E0", x"00000000");  
cpu_write_addr(x"E1", x"00000000");  
cpu_write_addr(x"E2", x"3F800000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"0000000B");
```

values. -- Setup Default World translation to screen coordinates matrix

```
cpu_write_addr(x"E0", x"3F800000");  
cpu_write_addr(x"E1", x"00000000");  
cpu_write_addr(x"E2", x"00000000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"00000000");
```

```
cpu_write_addr(x"E0", x"00000000");  
cpu_write_addr(x"E1", x"3F800000");  
cpu_write_addr(x"E2", x"00000000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"00000001");
```

```
cpu_write_addr(x"E0", x"00000000");  
cpu_write_addr(x"E1", x"00000000");  
cpu_write_addr(x"E2", x"3F800000");  
cpu_write_addr(x"E3", x"00000000");  
cpu_write_addr(x"E4", x"00000002");
```

```
cpu_write_addr(x"E0", x"00000000");  
cpu_write_addr(x"E1", x"00000000");  
cpu_write_addr(x"E2", x"00000000");  
cpu_write_addr(x"E3", x"3F800000");  
cpu_write_addr(x"E4", x"00000003");
```

-- Draw Cube

```
cpu_write_addr(x"F0", x"BE800000");  
cpu_write_addr(x"F1", x"BE800000");  
cpu_write_addr(x"F2", x"BE800000");  
cpu_write_addr(x"F3", x"3F800000");  
cpu_write_addr(x"F4", x"0003FFFF");  
cpu_write_addr(x"F5", x"00000001");  
cpu_write_addr(x"F0", x"BE800000");  
cpu_write_addr(x"F1", x"3E800000");  
cpu_write_addr(x"F2", x"BE800000");  
cpu_write_addr(x"F3", x"3F800000");
```



```
cpu_write_addr(x"F2", x"BF400000");
cpu_write_addr(x"F3", x"3F800000");
cpu_write_addr(x"F4", x"0003FFFF");
cpu_write_addr(x"F5", x"00000001");

cpu_write_addr(x"F0", x"3E800000");
cpu_write_addr(x"F1", x"BE800000");
cpu_write_addr(x"F2", x"BE800000");
cpu_write_addr(x"F3", x"3F800000");
cpu_write_addr(x"F4", x"0003FFFF");
cpu_write_addr(x"F5", x"00000001");
cpu_write_addr(x"F0", x"3E800000");
cpu_write_addr(x"F1", x"BE800000");
cpu_write_addr(x"F2", x"BF400000");
cpu_write_addr(x"F3", x"3F800000");
cpu_write_addr(x"F4", x"0003FFFF");
cpu_write_addr(x"F5", x"00000001");
end loop;

wait;

end process;

end architecture test;
```

APPENDIX D: C TESTCODE

This is the test code discussed in Section 4.5. The code provides initialization of the entire system. In addition the test code handles responding to the inputs of N64 controller, driving the objects into the graphics pipeline and updating the world and viewing translation matrices. The code is listed below.

```
/*
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 */

/*
 * Xilinx EDK 9.1.02 EDK_J_SP2.4
 *
 * This file is a sample test application
 *
 * This application is intended to test and/or illustrate some
 * functionality of your system. The contents of this file may
 * vary depending on the IP in your system and may use existing
 * IP driver functions. These drivers will be generated in your
 * XPS project when you run the "Generate Libraries" menu item
 * in XPS.
 *
 * Your XPS project directory is at:
 * C:\projects\gpu\xilinx\edk\microblaze\microblaze_0\
 */

// Located in: microblaze_0/include/xparameters.h
#include "xbasic_types.h"
#include "xparameters.h"
#include "xuartlite_1.h"
#include "xio.h"
#include "i2c.h"
#include "xenv.h"
#include "unistd.h"
```

```

#include "stdio.h"
#include "xutil.h"
#include "mb_interface.h"
#include "plb_regs_interface.h"
#include "xintc1.h"
#include "float.h"
#include "math.h"

//=====
#define MASTER_CLOCK 10000000 // 100Mhz
#define I2C_FREQUENCY 100000 // 100Khz
#define I2C_PRER_VALUE (MASTER_CLOCK / (5 * I2C_FREQUENCY)) - 1
#define I2C_WTIME 0x00080000 // ~5m

#define GPU_REG_BADDR XPAR_PLB_REGS_INTERFACE_0_MEMO_BASEADDR
#define N64_REG_BADDR XPAR_PLB_N64_CTRL_0_BASEADDR
#define I2C_BADDR XPAR_IIC_EEPROM_BASEADDR
#define VIDEO_I2C_BADDR XPAR_IIC_DVI_BASEADDR

#define EEPROM_ADDR 0xA0
#define DVI_ADDR 0xEC

#define DVI_RGB_BYPASS 0x21
#define DVI_TESTMODE 0x48
#define DVI_POWER_MNG 0x49
#define DVI_VERSION_ID 0x4A
#define DVI_DEVICE_ID 0x4B

#define GPU_ID GPU_REG_BADDR + 0x0
#define GPU_SCRATCH GPU_REG_BADDR + 0x4
#define GPU_DCM_STAT GPU_REG_BADDR + 0x8
#define GPU_CONTROL GPU_REG_BADDR + 0xC
#define GPU_BACKGROUND_COLOR GPU_REG_BADDR + 0x10
#define GPU_BACKGROUND_DEPTH GPU_REG_BADDR + 0x14
#define GPU_ZBT_MBOX_WRITE GPU_REG_BADDR + 0x40
#define GPU_ZBT_MBOX_READ GPU_REG_BADDR + 0x44
#define GPU_ZBT_MBOX_DATA0 GPU_REG_BADDR + 0x48
#define GPU_ZBT_MBOX_DATA1 GPU_REG_BADDR + 0x4C

#define GPU_CLIP_ZMAX GPU_REG_BADDR + 0x340

#define GPU_MATRIX_C0_DATA GPU_REG_BADDR + 0x380
#define GPU_MATRIX_C1_DATA GPU_REG_BADDR + 0x384
#define GPU_MATRIX_C2_DATA GPU_REG_BADDR + 0x388
#define GPU_MATRIX_C3_DATA GPU_REG_BADDR + 0x38C
#define GPU_MATRIX_WRITE GPU_REG_BADDR + 0x390

#define GPU_X_POINT GPU_REG_BADDR + 0x3C0
#define GPU_Y_POINT GPU_REG_BADDR + 0x3C4
#define GPU_Z_POINT GPU_REG_BADDR + 0x3C8
#define GPU_W_POINT GPU_REG_BADDR + 0x3CC
#define GPU_COLOR_POINT GPU_REG_BADDR + 0x3D0
#define GPU_POINT_TRIG GPU_REG_BADDR + 0x3D4

#define N64_CONTROL N64_REG_BADDR + 0x0
#define N64_STATUS N64_REG_BADDR + 0x4
#define N64_CW_RETURN N64_REG_BADDR + 0x8
#define N64_BS_RETURN N64_REG_BADDR + 0xc

//=====
const Xuint8 MICRON_AUX_REG = 0xF0;
const Xuint8 CAMERA_I2C_SLAVEADDR = 0xB8;
const float MOVE_INCR = 0.0025;

```



```

const float SCALE_INCR = 0.005;
const float ANGLE_INCR = 0.005 * M_PI;

// Define Globals.
volatile Xuint32 n64_button_reg;
volatile Xuint32 n64_button_reg_prev;
volatile Xuint32 scene_update;

void init_peripherals() {

    Xuint8          i2c_read_byte;
    Xuint32         gpu_read_word;
    volatile float  *gpu_float_ptr;
    volatile float  gpu_float;
    volatile int    *gpu_int_ptr;
    volatile float  gpu_matrix[4][4];
    int x, y, i, j;

    // Initialize the DVI I2C core.
    xil_printf("Initializing DVI DAC I2C Bus... \r\n");

I2C_MASTER_initCore(VIDEO_I2C_BADDR, I2C_PRER_VALUE, I2C_WTIME, MICRON_AUX_REG);

    // Read out DVI Version ID.
    i2c_read_byte =
    I2C_MASTER_readByte(VIDEO_I2C_BADDR, DVI_ADDR, DVI_VERSION_ID);
    xil_printf("DVI DAC VERSION ID Reg = 0x%x \n\r", i2c_read_byte);

    // Read out DVI Device ID.
    i2c_read_byte =
    I2C_MASTER_readByte(VIDEO_I2C_BADDR, DVI_ADDR, DVI_DEVICE_ID);
    xil_printf("DVI DAC DEVICE ID Reg = 0x%x \n\r", i2c_read_byte);

    // Setup RGB Bypass mode.
    print("Enabling RGB bypass mode \n\r");
    I2C_MASTER_writeByte(VIDEO_I2C_BADDR, DVI_ADDR, DVI_RGB_BYPASS, 0x9);
    i2c_read_byte =
    I2C_MASTER_readByte(VIDEO_I2C_BADDR, DVI_ADDR, DVI_RGB_BYPASS);
    xil_printf("DVI RGB Bypass Reg = 0x%x \n\r", i2c_read_byte);

    // Power on DAC.
    print("Powering up DAC \n\r");
    I2C_MASTER_writeByte(VIDEO_I2C_BADDR, DVI_ADDR, DVI_POWER_MNG, 0x0);
    i2c_read_byte =
    I2C_MASTER_readByte(VIDEO_I2C_BADDR, DVI_ADDR, DVI_POWER_MNG);
    xil_printf("DVI Power Managment Reg = 0x%x \n\r", i2c_read_byte);

    // Check GPU id.
    gpu_read_word = XIo_In32(GPU_ID);
    xil_printf("GPU ID Reg = 0x%x \n\r", gpu_read_word);

    // Determine if controller is present
    gpu_read_word = XIo_In32(N64_STATUS);
    if ((gpu_read_word & 0x00000002) == 0) {
        xil_printf("N64 Controller Not Present\n\r");
    } else {
        xil_printf("N64 Controller Present\n\r");
    }

    // Clear out Memory
    print("Clearing Memory \n\r");
    XIo_Out32(GPU_ZBT_MBOX_DATA0, 0x00010A35);
    XIo_Out32(GPU_ZBT_MBOX_DATA1, 0x0);

```

```

for (x=0; x<320; x++) {
  for (y=0; y<240; y++) {
    XIo_Out32(GPU_ZBT_MBOX_WRITE, ((x+(y*320))*4));
    gpu_read_word = XIo_In32(GPU_DCM_STAT);
  }
}

// Set background color.
print("Setting Background Color\n\r");
XIo_Out32(GPU_BACKGROUND_COLOR, 0x0000003F);
gpu_read_word = XIo_In32(GPU_BACKGROUND_COLOR);
xil_printf("GPU Background Color = 0x%x \n\r", gpu_read_word);

print("Setting Background Color\n\r");
XIo_Out32(GPU_BACKGROUND_DEPTH, 0x00000000);
gpu_read_word = XIo_In32(GPU_BACKGROUND_DEPTH);
xil_printf("GPU Background Depth = 0x%x \n\r", gpu_read_word);

// Set Clipping Registers.
print("Setting ZMax Clipping Plane\n\r");
XIo_Out32(GPU_CLIP_ZMAX, 0xBDCCCC);
gpu_read_word = XIo_In32(GPU_CLIP_ZMAX);
xil_printf("GPU Background Color = 0x%x \n\r", gpu_read_word);

// Set Viewport to Screen Translation Matrix.
gpu_matrix[0][0] = 159.5; gpu_matrix[0][1] = 0.0;   gpu_matrix[0][2] = 0.0;
gpu_matrix[0][3] = 159.5;
gpu_matrix[1][0] = 0.0;   gpu_matrix[1][1] = 119.5; gpu_matrix[1][2] = 0.0;
gpu_matrix[1][3] = 119.5;
gpu_matrix[2][0] = 0.0;   gpu_matrix[2][1] = 0.0;   gpu_matrix[2][2] = 1.0;
gpu_matrix[2][3] = 0.0;
gpu_matrix[3][0] = 0.0;   gpu_matrix[3][1] = 0.0;   gpu_matrix[3][2] = 0.0;
gpu_matrix[3][3] = 1.0;

// Write transformation matrix to hardware.
xil_printf("Setting up Screen Translation Matrix\n\r");
gpu_float_ptr = &gpu_float;
for (i=0; i<=3; i++) {
  for (j=0; j<=3; j++) {
    gpu_float = gpu_matrix[i][j];
    gpu_int_ptr = (int*)&gpu_float;
    if (j==0) {
      XIo_Out32(GPU_MATRIX_C0_DATA, *gpu_int_ptr);
    } else if (j==1) {
      XIo_Out32(GPU_MATRIX_C1_DATA, *gpu_int_ptr);
    } else if (j==2) {
      XIo_Out32(GPU_MATRIX_C2_DATA, *gpu_int_ptr);
    } else {
      XIo_Out32(GPU_MATRIX_C3_DATA, *gpu_int_ptr);
      XIo_Out32(GPU_MATRIX_WRITE, (0xc | i));
    }
  }
}

// Set scene semaphore
scene_update = 0;

// Enable GPU/VGA Logic
print("Enabling GPU Logic \n\r");
XIo_Out32(GPU_CONTROL, 0x3);
gpu_read_word = XIo_In32(GPU_CONTROL);
xil_printf("GPU Control Reg = 0x%x \n\r", gpu_read_word);

```

```

}

void eof_intr_handler(void * baseaddr_p) {

    Xuint32 ctrl_read;

    Xuint32 baseaddr;
    Xuint32 IntrStatus;
    Xuint32 IpStatus;
    baseaddr = (Xuint32) baseaddr_p;

    // Get status from Device Interrupt Status Register.
    IntrStatus
    PLB_REGS_INTERFACE_mReadReg(baseaddr, PLB_REGS_INTERFACE_INTR_DISR_OFFSET); =

    // Determine if controller is present.
    ctrl_read = XIo_In32(N64_STATUS);
    if ((ctrl_read & 0x00000002) == 0x00000002) {

        // Trigger a read of the controller's status registers.
        XIo_Out32(N64_CONTROL, 0x00000004);

        // Wait until the state machine is no longer busy.
        ctrl_read = 0;
        while ((ctrl_read & 0x00000001) == 0) {
            ctrl_read = XIo_In32(N64_STATUS);
        }

        // Read status registers.
        ctrl_read = XIo_In32(N64_BS_RETURN);

        n64_button_reg_prev = n64_button_reg;
        n64_button_reg      = ctrl_read;

    }

    // Set scene update variable.
    scene_update = 1;

    // Verify the source of the interrupt is the user logic and clear the
    // source by toggle write back to the IP ISR register.
    if ( (IntrStatus & INTR_IPIR_MASK) == INTR_IPIR_MASK ) {
        IpStatus
        PLB_REGS_INTERFACE_mReadReg(baseaddr, PLB_REGS_INTERFACE_INTR_IPIR_OFFSET); =
        PLB_REGS_INTERFACE_mWriteReg(baseaddr, PLB_REGS_INTERFACE_INTR_IPIR_OFFSET,
        IpStatus);
    }

}

//=====
int main (void) {

    // Object 1
    volatile float x0[8], x1[5], vcx, vax, vex, cx[2], tx[2], sx[2], ax[2];
    volatile float y0[8], y1[5], vcy, vay, vey, cy[2], ty[2], sy[2], ay[2];
    volatile float z0[8], z1[5], vcz, vaz, vez, cz[2], tz[2], sz[2], az[2];
    volatile int   color[2];
    volatile float angle, scale;
    volatile float x0_float, y0_float, z0_float;

```

```

volatile float *x0_float_ptr, *y0_float_ptr, *z0_float_ptr;
volatile int *x0_int_ptr, *y0_int_ptr, *z0_int_ptr;
volatile float x1_float, y1_float, z1_float;
volatile float *x1_float_ptr, *y1_float_ptr, *z1_float_ptr;
volatile int *x1_int_ptr, *y1_int_ptr, *z1_int_ptr;
volatile float matrix[4][4];
volatile float *matrix_float_ptr;
volatile float matrix_float;
volatile int *matrix_int_ptr;
volatile float cosz, sinz, cosy, siny, cosx, sinx;
volatile float active_object = -1;
int i, j;
int k = 0;

/*
 * Enable and initialize cache
 */
#if XPAR_MICROBLAZE_0_USE_ICACHE
microblaze_init_cache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
microblaze_enable_cache();
#endif

#if XPAR_MICROBLAZE_0_USE_DCACHE
microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
microblaze_enable_dcache();
#endif

print("-- Powering Up GPU! --\r\n");
init_peripherals();

xil_printf("Initializing Interrupts \n\r");

// Register Handler.
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
XPAR_XPS_INTC_0_PLB_REGS_INTERFACE_0_IP2INTC_IRPT_INTR, (XInterruptHandler) eof_
_intr_handler, (void *)XPAR_PLB_REGS_INTERFACE_0_BASEADDR);

// Start the interrupt controller
XIntc_MasterEnable(XPAR_XPS_INTC_0_BASEADDR);

// Enable requests in the interrupt controller
XIntc_EnableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_PLB_REGS_INTERFACE_0_IP2INTC_
IRPT_MASK);

// Enable Interrupts on PLB register controller.
PLB_REGS_INTERFACE_EnableInterrupt((Xuint32
*)XPAR_PLB_REGS_INTERFACE_0_BASEADDR);

// Enable interrupts.
microblaze_enable_interrupts();

// Define objects local coordinate system location
x0[0] = -0.1; y0[0] = -0.1; z0[0] = -0.1;
x0[1] = -0.1; y0[1] = 0.1; z0[1] = -0.1;
x0[2] = 0.1; y0[2] = 0.1; z0[2] = -0.1;
x0[3] = 0.1; y0[3] = -0.1; z0[3] = -0.1;

x0[4] = -0.1; y0[4] = -0.1; z0[4] = 0.1;
x0[5] = -0.1; y0[5] = 0.1; z0[5] = 0.1;
x0[6] = 0.1; y0[6] = 0.1; z0[6] = 0.1;
x0[7] = 0.1; y0[7] = -0.1; z0[7] = 0.1;

cx[0] = 0.0; cy[0] = 0.0; cz[0] = 0.00;

```

```

tx[0] = 0.0; ty[0] = 0.0; tz[0] = -0.35;
ax[0] = 0.0; ay[0] = 0.0; az[0] = 0.00;
sx[0] = 1.0; sy[0] = 1.0; sz[0] = 1.00;
color[0] = 0x0003FFFF;

x1[0] = -0.1; y1[0] = -0.1; z1[0] = -0.1;
x1[1] = 0.1; y1[1] = -0.1; z1[1] = -0.1;
x1[2] = 0.1; y1[2] = -0.1; z1[2] = 0.1;
x1[3] = -0.1; y1[3] = -0.1; z1[3] = 0.1;
x1[4] = 0.0; y1[4] = 0.1; z1[4] = 0.0;

cx[1] = 0.0; cy[1] = 0.0; cz[1] = 0.00;
tx[1] = 0.0; ty[1] = 0.0; tz[1] = -0.75;
ax[1] = 0.0; ay[1] = 0.0; az[1] = 0.00;
sx[1] = 1.0; sy[1] = 1.0; sz[1] = 1.00;
color[1] = 0x0003FFFF;

// Viewing Parameters.
vcx = vcy = vcz = 0.0;
vax = vay = vaz = 0.0;
vex = vey = vez = 0.0;

while (TRUE) {
    if (scene_update == 1) {

        if (active_object == -1) {

            // Update View Parameters based on button presses.
            if ((n64_button_reg & 0x00280000) == 0x00080000) {
                vcz = vcz - MOVE_INCR;
            }
            if ((n64_button_reg & 0x00240000) == 0x00040000) {
                vcz = vcz + MOVE_INCR;
            }
            if ((n64_button_reg & 0x08200000) == 0x08000000) {
                vcy = vcy - MOVE_INCR;
            }
            if ((n64_button_reg & 0x04200000) == 0x04000000) {
                vcy = vcy + MOVE_INCR;
            }
            if ((n64_button_reg & 0x02200000) == 0x02000000) {
                vcx = vcx - MOVE_INCR;
            }
            if ((n64_button_reg & 0x01200000) == 0x01000000) {
                vcx = vcx + MOVE_INCR;
            }

            if ((n64_button_reg & 0x00220000) == 0x00220000) {
                vaz = vaz - ANGLE_INCR;
            }
            if ((n64_button_reg & 0x00210000) == 0x00210000) {
                vaz = vaz + ANGLE_INCR;
            }
            if ((n64_button_reg & 0x08200000) == 0x08200000) {
                vax = vax + ANGLE_INCR;
            }
            if ((n64_button_reg & 0x04200000) == 0x04200000) {
                vax = vax - ANGLE_INCR;
            }
            if ((n64_button_reg & 0x02200000) == 0x02200000) {
                vay = vay - ANGLE_INCR;
            }
            if ((n64_button_reg & 0x01200000) == 0x01200000) {
                vay = vay + ANGLE_INCR;
            }
        }
    }
}

```

```

}

if ((n64_button_reg & 0x10000000) == 0x10000000) {
    vcx = vcy = vcz = 0.0;
    vax = vay = vaz = 0.0;
    vex = vey = vez = 0.0;
}
} else if (active_object == 0) {

    // Update View Parameters based on button presses.
    if ((n64_button_reg & 0x00280000) == 0x00080000) {
        tz[0] = tz[0] - MOVE_INCR;
    }
    if ((n64_button_reg & 0x00240000) == 0x00040000) {
        tz[0] = tz[0] + MOVE_INCR;
    }
    if ((n64_button_reg & 0x08200000) == 0x08000000) {
        ty[0] = ty[0] + MOVE_INCR;
    }
    if ((n64_button_reg & 0x04200000) == 0x04000000) {
        ty[0] = ty[0] - MOVE_INCR;
    }
    if ((n64_button_reg & 0x02200000) == 0x02000000) {
        tx[0] = tx[0] + MOVE_INCR;
    }
    if ((n64_button_reg & 0x01200000) == 0x01000000) {
        tx[0] = tx[0] - MOVE_INCR;
    }

    if ((n64_button_reg & 0x40200000) == 0x40000000) {
        sx[0] = sy[0] = sz[0] = sx[0] - SCALE_INCR;
    }
    if ((n64_button_reg & 0x80200000) == 0x80000000) {
        sx[0] = sy[0] = sz[0] = sx[0] + SCALE_INCR;
    }

    if ((n64_button_reg & 0x10000000) == 0x10000000) {
        tx[0] = 0.0; ty[0] = 0.0; tz[0] = -0.35;
        ax[0] = 0.0; ay[0] = 0.0; az[0] = 0.0;
        sx[0] = 1.0; sy[0] = 1.0; sz[0] = 1.0;
    }
}

} else if (active_object == 1) {

    // Update View Parameters based on button presses.
    if ((n64_button_reg & 0x00280000) == 0x00080000) {
        tz[1] = tz[1] - MOVE_INCR;
    }
    if ((n64_button_reg & 0x00240000) == 0x00040000) {
        tz[1] = tz[1] + MOVE_INCR;
    }
    if ((n64_button_reg & 0x08200000) == 0x08000000) {
        ty[1] = ty[1] + MOVE_INCR;
    }
    if ((n64_button_reg & 0x04200000) == 0x04000000) {
        ty[1] = ty[1] - MOVE_INCR;
    }
    if ((n64_button_reg & 0x02200000) == 0x02000000) {
        tx[1] = tx[1] + MOVE_INCR;
    }
    if ((n64_button_reg & 0x01200000) == 0x01000000) {
        tx[1] = tx[1] - MOVE_INCR;
    }
}

```

```

    if ((n64_button_reg & 0x40200000) == 0x40000000) {
        sx[1] = sy[1] = sz[1] = sx[1] - SCALE_INCR;
    }
    if ((n64_button_reg & 0x80200000) == 0x80000000) {
        sx[1] = sy[1] = sz[1] = sx[1] + SCALE_INCR;
    }

    if ((n64_button_reg & 0x10000000) == 0x10000000) {
        tx[1] = 0.0; ty[1] = 0.0; tz[1] = -0.75;
        ax[1] = 0.0; ay[1] = 0.0; az[1] = 0.0;
        sx[1] = 1.0; sy[1] = 1.0; sz[1] = 1.0;
    }
}

if (((n64_button_reg & 0x00100000) == 0x00100000) &&
    ((n64_button_reg_prev & 0x00100000) == 0x00000000)) {
if (active_object == -1) {
    active_object = active_object + 1;
    color[0] = 0x00FC0;
    color[1] = 0x3FFFF;
} else if (active_object == 0) {
    active_object = active_object + 1;
    color[0] = 0x3FFFF;
    color[1] = 0x00FC0;
} else {
    active_object = -1;
    color[0] = 0x3FFFF;
    color[1] = 0x3FFFF;
}
}

}

// Set Translation Matrix to Identity.
cosx = cos(-vax); cosy = cos(-vay); cosz = cos(-vaz);
sinx = sin(-vax); siny = sin(-vay); sinz = sin(-vaz);
matrix[0][0] = cosy*cosz;
matrix[0][1] = cosy*sinz;
matrix[0][2] = -siny;
matrix[0][3] = -vcx * cosy * cosz - vcy * cosy * sinz + vcz * siny;

matrix[1][0] = -cosx * sinz + sinx * siny * cosz;
matrix[1][1] = cosx * cosz + sinx * siny * sinz;
matrix[1][2] = sinx*cosy;
matrix[1][3] = -vcy * cosx * cosz + vcx * cosx * sinz - sinx * (vcz *
cosy + vcy * siny * sinz) - vcx * sinx * siny * cosz;

matrix[2][0] = cosx * siny * cosz + sinx * sinz;
matrix[2][1] = cosx * siny * sinz - sinx * cosz;
matrix[2][2] = cosx*cosy;
matrix[2][3] = -vcz * cosx * cosy - vcx * cosx * siny * cosz - vcy *
cosx * siny * sinz + vcy * sinx * cosz - vcx * sinx * sinz;

matrix[3][0] = 0.0;
matrix[3][1] = 0.0;
matrix[3][2] = 0.0;
matrix[3][3] = 1.0;

// Write world to view transformation matrix to hardware.
matrix_float_ptr = &matrix_float;
for (i=0; i<=3; i++) {
    for (j=0; j<=3; j++) {
        matrix_float = matrix[i][j];
    }
}

```

```

matrix_int_ptr = (int*)&matrix_float;
if (j==0) {
    XIo_Out32(GPU_MATRIX_C0_DATA, *matrix_int_ptr);
} else if (j==1) {
    XIo_Out32(GPU_MATRIX_C1_DATA, *matrix_int_ptr);
} else if (j==2) {
    XIo_Out32(GPU_MATRIX_C2_DATA, *matrix_int_ptr);
} else {
    XIo_Out32(GPU_MATRIX_C3_DATA, *matrix_int_ptr);
    XIo_Out32(GPU_MATRIX_WRITE, (0x4 | i));
}
}
}

// Set Translation Matrix to Identity.
matrix[0][0] = 1.0; matrix[0][1] = 0.0; matrix[0][2] = 0.0;
matrix[0][3] = -vex;
matrix[1][0] = 0.0; matrix[1][1] = 1.0; matrix[1][2] = 0.0;
matrix[1][3] = -vey;
matrix[2][0] = 0.0; matrix[2][1] = 0.0; matrix[2][2] = 2.0;
matrix[2][3] = 0.0;
matrix[3][0] = 0.0; matrix[3][1] = 0.0; matrix[3][2] = 1.0;
matrix[3][3] = 0.0;

// Write view to projection transformation matrix to hardware.
//matrix_float_ptr = &matrix_float;
for (i=0; i<=3; i++) {
    for (j=0; j<=3; j++) {
        matrix_float = matrix[i][j];
        matrix_int_ptr = (int*)&matrix_float;
        if (j==0) {
            XIo_Out32(GPU_MATRIX_C0_DATA, *matrix_int_ptr);
        } else if (j==1) {
            XIo_Out32(GPU_MATRIX_C1_DATA, *matrix_int_ptr);
        } else if (j==2) {
            XIo_Out32(GPU_MATRIX_C2_DATA, *matrix_int_ptr);
        } else {
            XIo_Out32(GPU_MATRIX_C3_DATA, *matrix_int_ptr);
            XIo_Out32(GPU_MATRIX_WRITE, (0x8 | i));
        }
    }
}

// Clear Semaphore
scene_update = 0;

// Configure Local to World Translation Matrix
matrix[0][0] = sx[0]; matrix[0][1] = 0.0; matrix[0][2] = 0.0;
matrix[0][3] = tx[0];
matrix[1][0] = 0.0; matrix[1][1] = sy[0]; matrix[1][2] = 0.0;
matrix[1][3] = ty[0];
matrix[2][0] = 0.0; matrix[2][1] = 0.0; matrix[2][2] = sz[0];
matrix[2][3] = tz[0];
matrix[3][0] = 0.0; matrix[3][1] = 0.0; matrix[3][2] = 0.0;
matrix[3][3] = 1.0;

// Write local to world transformation matrix to hardware.
matrix_float_ptr = &matrix_float;
for (i=0; i<=3; i++) {
    for (j=0; j<=3; j++) {
        matrix_float = matrix[i][j];
        matrix_int_ptr = (int*)&matrix_float;
        if (j==0) {
            XIo_Out32(GPU_MATRIX_C0_DATA, *matrix_int_ptr);

```



```

    } else if (j==1) {
        XIo_Out32(GPU_MATRIX_C1_DATA, *matrix_int_ptr);
    } else if (j==2) {
        XIo_Out32(GPU_MATRIX_C2_DATA, *matrix_int_ptr);
    } else {
        XIo_Out32(GPU_MATRIX_C3_DATA, *matrix_int_ptr);
        XIo_Out32(GPU_MATRIX_WRITE, (0x0 | i));
    }
}
}
}

```

```

// Set Color and W value.
XIo_Out32(GPU_COLOR_POINT, color[0]);
XIo_Out32(GPU_W_POINT, 0x3F800000);

```

```

x0_float_ptr = &x0_float;
y0_float_ptr = &y0_float;
z0_float_ptr = &z0_float;
x1_float_ptr = &x1_float;
y1_float_ptr = &y1_float;
z1_float_ptr = &z1_float;
for (i=0; i<4; i++) {

    // Write Line
    x0_float = x0[i]; x0_int_ptr = (int*)&x0_float;
    y0_float = y0[i]; y0_int_ptr = (int*)&y0_float;
    z0_float = z0[i]; z0_int_ptr = (int*)&z0_float;
    XIo_Out32(GPU_X_POINT, *x0_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y0_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z0_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
    if (i == 3) {
        x1_float = x0[0]; x1_int_ptr = (int*)&x1_float;
        y1_float = y0[0]; y1_int_ptr = (int*)&y1_float;
        z1_float = z0[0]; z1_int_ptr = (int*)&z1_float;
    } else {
        x1_float = x0[i+1]; x1_int_ptr = (int*)&x1_float;
        y1_float = y0[i+1]; y1_int_ptr = (int*)&y1_float;
        z1_float = z0[i+1]; z1_int_ptr = (int*)&z1_float;
    }
    XIo_Out32(GPU_X_POINT, *x1_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y1_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z1_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
}
}

```

```

for (i=4; i<8; i++) {

    // Write Line
    x0_float = x0[i]; x0_int_ptr = (int*)&x0_float;
    y0_float = y0[i]; y0_int_ptr = (int*)&y0_float;
    z0_float = z0[i]; z0_int_ptr = (int*)&z0_float;
    XIo_Out32(GPU_X_POINT, *x0_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y0_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z0_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
    if (i == 7) {
        x1_float = x0[4]; x1_int_ptr = (int*)&x1_float;
        y1_float = y0[4]; y1_int_ptr = (int*)&y1_float;
        z1_float = z0[4]; z1_int_ptr = (int*)&z1_float;
    } else {
        x1_float = x0[i+1]; x1_int_ptr = (int*)&x1_float;
        y1_float = y0[i+1]; y1_int_ptr = (int*)&y1_float;
    }
}
}

```

```

    z1_float = z0[i+1]; z1_int_ptr = (int*)&z1_float;
}
XIo_Out32(GPU_X_POINT , *x1_int_ptr);
XIo_Out32(GPU_Y_POINT , *y1_int_ptr);
XIo_Out32(GPU_Z_POINT , *z1_int_ptr);
XIo_Out32(GPU_POINT_TRIG, 0x0);

}

for (i=0; i<4; i++) {

    // Write Line
    x0_float = x0[i]; x0_int_ptr = (int*)&x0_float;
    y0_float = y0[i]; y0_int_ptr = (int*)&y0_float;
    z0_float = z0[i]; z0_int_ptr = (int*)&z0_float;
    XIo_Out32(GPU_X_POINT , *x0_int_ptr);
    XIo_Out32(GPU_Y_POINT , *y0_int_ptr);
    XIo_Out32(GPU_Z_POINT , *z0_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
    x1_float = x0[i+4]; x1_int_ptr = (int*)&x1_float;
    y1_float = y0[i+4]; y1_int_ptr = (int*)&y1_float;
    z1_float = z0[i+4]; z1_int_ptr = (int*)&z1_float;
    XIo_Out32(GPU_X_POINT , *x1_int_ptr);
    XIo_Out32(GPU_Y_POINT , *y1_int_ptr);
    XIo_Out32(GPU_Z_POINT , *z1_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);

}

// Wait for a little, this is a hack.
//for (i=0; i<50; i++) {
//    k=k+1;
//}

// Configure Local to World Translation Matrix
matrix[0][0] = sx[1]; matrix[0][1] = 0.0; matrix[0][2] = 0.0;
matrix[0][3] = tx[1];
matrix[1][0] = 0.0; matrix[1][1] = sy[1]; matrix[1][2] = 0.0;
matrix[1][3] = ty[1];
matrix[2][0] = 0.0; matrix[2][1] = 0.0; matrix[2][2] = sz[1];
matrix[2][3] = tz[1];
matrix[3][0] = 0.0; matrix[3][1] = 0.0; matrix[3][2] = 0.0;
matrix[3][3] = 1.0;

// Write local to world transformation matrix to hardware.
matrix_float_ptr = &matrix_float;
for (i=0; i<=3; i++) {
    for (j=0; j<=3; j++) {
        matrix_float = matrix[i][j];
        matrix_int_ptr = (int*)&matrix_float;
        if (j==0) {
            XIo_Out32(GPU_MATRIX_C0_DATA, *matrix_int_ptr);
        } else if (j==1) {
            XIo_Out32(GPU_MATRIX_C1_DATA, *matrix_int_ptr);
        } else if (j==2) {
            XIo_Out32(GPU_MATRIX_C2_DATA, *matrix_int_ptr);
        } else {
            XIo_Out32(GPU_MATRIX_C3_DATA, *matrix_int_ptr);
            XIo_Out32(GPU_MATRIX_WRITE, (0x0 | i));
        }
    }
}
}

```

```

// Set Color and W value.
XIo_Out32(GPU_COLOR_POINT, color[1]);
XIo_Out32(GPU_W_POINT, 0x3F800000);

for (i=0; i<4; i++) {
    // Write Line
    x0_float = x1[i]; x0_int_ptr = (int*)&x0_float;
    y0_float = y1[i]; y0_int_ptr = (int*)&y0_float;
    z0_float = z1[i]; z0_int_ptr = (int*)&z0_float;
    XIo_Out32(GPU_X_POINT, *x0_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y0_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z0_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
    if (i == 3) {
        x1_float = x1[0]; x1_int_ptr = (int*)&x1_float;
        y1_float = y1[0]; y1_int_ptr = (int*)&y1_float;
        z1_float = z1[0]; z1_int_ptr = (int*)&z1_float;
    } else {
        x1_float = x1[i+1]; x1_int_ptr = (int*)&x1_float;
        y1_float = y1[i+1]; y1_int_ptr = (int*)&y1_float;
        z1_float = z1[i+1]; z1_int_ptr = (int*)&z1_float;
    }
    XIo_Out32(GPU_X_POINT, *x1_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y1_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z1_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
}

for (i=0; i<4; i++) {
    // Write Line
    x0_float = x1[i]; x0_int_ptr = (int*)&x0_float;
    y0_float = y1[i]; y0_int_ptr = (int*)&y0_float;
    z0_float = z1[i]; z0_int_ptr = (int*)&z0_float;
    XIo_Out32(GPU_X_POINT, *x0_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y0_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z0_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
    x1_float = x1[4]; x1_int_ptr = (int*)&x1_float;
    y1_float = y1[4]; y1_int_ptr = (int*)&y1_float;
    z1_float = z1[4]; z1_int_ptr = (int*)&z1_float;
    XIo_Out32(GPU_X_POINT, *x1_int_ptr);
    XIo_Out32(GPU_Y_POINT, *y1_int_ptr);
    XIo_Out32(GPU_Z_POINT, *z1_int_ptr);
    XIo_Out32(GPU_POINT_TRIG, 0x0);
}
}

/*
 * Disable cache and reinitialize it so that other
 * applications can be run with no problems
 */
#if XPAR_MICROBLAZE_0_USE_DCACHE
    microblaze_disable_dcache();
    microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
#endif

#if XPAR_MICROBLAZE_0_USE_ICACHE
    microblaze_disable_icache();
    microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
#endif

```

```
    print("-- Exiting main() --\r\n");  
    return 0;  
}
```

BIBLIOGRAPHY

1. **Watt, Alan.** *3D Computer Graphics*. s.l. : Addison-Wesley Publishing, 2000.
2. **NVIDIA Corporation.** Geforce 8800 GPU Architecture Technical Brief. [Online] November 2006. http://www.nvidia.com/page/8800_tech_briefs.html.
3. *Dense Matrix-Vector Multiplication on the CUDA Architecture.* **Fujimoto, Noriyuki.** s.l. : Parallel Processing Letters, 2008.
4. **Xilinx, Inc.** Virtex-5 Family Overview. [Online] September 23, 2008. <http://www.xilinx.com>.
5. Wikipedia Article "Raster Graphics". *Wikipedia*. [Online] [Cited: September 10, 2008.] http://en.wikipedia.org/wiki/Raster_graphics.
6. **Hecker, Stephen Andrilli and David.** *Elementary Linear Algebra*. s.l. : PWS Publishing Company, 1993.
7. **James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes.** *Computer Graphics Principles and Practice*. s.l. : Addison-Wesley Publishing Company, Inc, 1996.
8. **Sproull, W. Newman and R.** *Principles of Interactive Computer Graphics*. New York : McGraw-Hill, 1979.
9. *Algorithm for computer control of a digital plotter.* **Bresenham, J. E.** 1, s.l. : IBM Systems Journal, 1965, Vol. 4.

10. Wikipedia Article "Bresenham's Line Algorithm". *Wikipedia*. [Online] [Cited: September 10, 2008.] http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm.
11. **Flannagan, Colin**. Website "The Bresenham Line-Drawing Algorithm". [Online] [Cited: September 10, 2008.] <http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html>.
12. **Xilinx, Inc.** Virtex-5 FPGA Datasheets. [Online] September 23, 2008. <http://www.xilinx.com>.
13. **Xilinx, Inc.** ML505/ML506/ML507. [Online] November 10, 2008. <http://www.xilinx.com>.
14. *IEEE Standard for Binary Floating-Point Arithmetic*. **IEEE Computer Society**. 1985. IEEE Std. 754-1985.
15. **Xilinx Inc.** Coregen IP Generator Documentation. [Online] April 2008. <http://www.xilinx.com>.
16. **Xilinx, Inc.** Floating-Point Operator v4.0. [Online] April 25, 2008. <http://www.xilinx.com>.
17. **Xilinx, Inc.** Microblaze Processor Reference Guide. [Online] <http://www.xilinx.com>.
18. **Xilinx, Inc.** Multi-Port Memory Controller (MPMC) (v3.00.b). [Online] November 8, 2007. <http://www.xilinx.com>.
19. **Xilinx, Inc.** Processor Local Bus (PLB) v4.6 (v1.00a). [Online] [Cited: August 9, 2007.] <http://www.xilinx.com>.
20. Wikipedia Article "Nintendo 64 Controller". *Wikipedia*. [Online] [Cited: September 10, 2008.] http://en.wikipedia.org/wiki/Nintendo_64_controller.