TEAM START BUTTON

{kke, teckl, kylim, joomayt} @ andrew.cmu.edu
Website URL: http://start-button.blogspot.com/

OPENGL GRAPHICS ACCELERATOR
FINAL PROJECT REPORT

Kenneth Eng
Teck Hua Lee
Ken Yu Lim
Joomay Tan

Table of Contents

1. INTRODUCTION..... 3

2. OPENGL..... 4

2.1 OPENGL LIBRARY
2.2 SUPPORTED OPENGL FUNCTIONS

3. GPU'S INTRUCTION SET ARCHITECTURE 5

3.1 ISA SPECIFICATION
3.2 ISA DISCUSSION
3.3 FIXED POINT

4. FETCH UNIT..... 8

4.1 SERVER ON POWERPC CORE
4.2 INSTRUCTION BRAM

5. MATRICES & TRANSFORMATIONS..... 11

5.1 MATRIX STACK UPDATE
5.2 COORDINATE TRANSFORMATION PIPELINE

6. RASTERIZATION..... 15

6.1 ALGORITHM
6.2 COLOR INTERPOLATION

7. FRAME BUFFER 21

8. DESIGN APPROACH ……....26

8.1 DESIGN PARTINIONING
8.2 DESIGN METHODOLOGY
8.3 TESTING & VERIFICATION METHODOLOGY

9. STATUS & FUTURE WORK ……....28

10. LESSONS LEARNED ……...29

11. INDIVIDUAL COMMENTS ……31

12. CITATION OF RESOURCES USED ……36

1) INTRODUCTION

OpenGL (Open Graphics Library) is the industry standard for high performance graphics which enables rendering of 3D and 2D computer graphics. It is widely used in the production of video games, as well as in other visualization purposes. It defines a specification from which hardware makers implement their designs to perform OpenGL routines through hardware acceleration. There are over 250 different function calls which process primitives such as points, lines, and polygons into pixels for display.

In this project, we designed a simple graphics pipeline from scratch and implemented it on a Xilinx XC2VP30 FPGA. We designed the overall architecture of the pipeline and implemented all the functional units in Verilog. We focused on the following modules in the pipeline: Fetch and Decode unit, Coordinate Transformation unit, Rasterization unit, and the frame buffer. Instruction and data are sent from a connecting workstation to the FPGA board using TCP/IP. The output from the FPGA is displayed on a Sony 640x480 VGA. Figure 1 shows the high-level system diagram of our design.

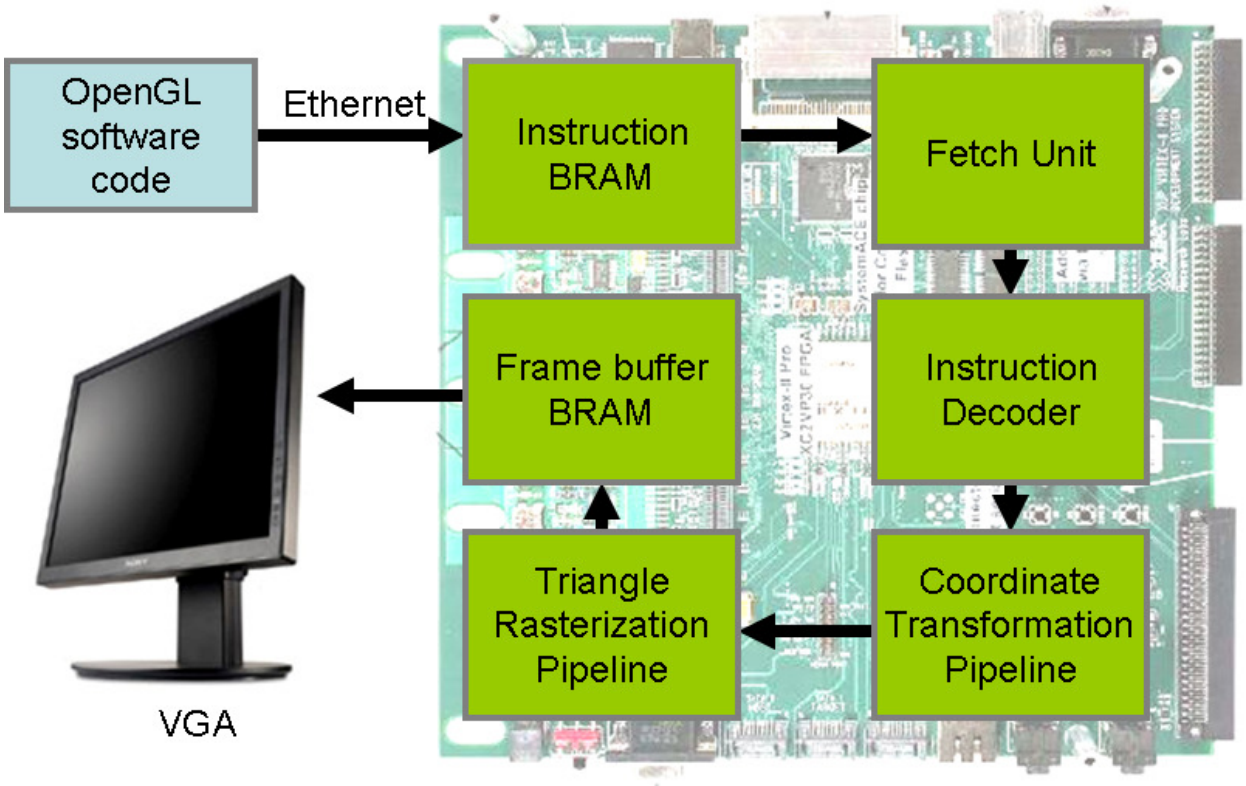


Figure 1: High-level System Diagram

This report details the hardware design and specification that we implemented in one semester.

2) OPENGL

2.1 OpenGL Library

Our OpenGL Library is adapted from GLSim, an OpenGL simulator from Stanford University (http://graphics.stanford.edu/courses/cs448a-01-fall/glsim.html).

GLSim was chosen because it already has a well defined framework to parse OpenGL functions. This library compiles into libGL.so which replaces the default Mesa OpenGL library. We replaced the software implementation of the graphics pipeline in the simulator with skeleton functions for every OpenGL routine. These functions map the OpenGL calls into our GPU's ISA. The byte code will then be reordered if necessary and sent to the FPGA board through TCP/IP.

2.2 Supported OpenGL Functions

Due to time and resource constraints, only the following subset of the OpenGL specification is supported.

Primitives
void glBegin (GLenum mode)
void glEnd (void)
void glVertex (TYPE x, TYPE y, TYPE z, TYPE w)
void glColor (TYPE red, TYPE green, TYPE blue, TYPE alpha)

Transform
void glLoadIdentity (void)
void glLoadMatrix (const TYPE *m)
void glMatrixMode (GLenum mode)
void glMultMatrix (const TYPE *m)
void glPopMatrix (void)
void glPushMatrix (void)
void glRotate (TYPE angle, TYPE x, TYPE y, TYPE z)
void glScale (TYPE x, TYPE y, TYPE z)
void glTranslate (TYPE x, TYPE y, TYPE z)
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height)

Arrays
void glVertexPointer (GLint size, GLenum type, GLsizei stride, const GLvoid *pointer)

All OpenGL Arrays routines are expanded by our OpenGL Library into Primitives routines by our OpenGL library. This is due to the limitations of the fetch unit to which are detailed in Section 4.

3) GPU's INSTRUCTION SET ARCHITECTURE

3.1 ISA Specification

The ISA defines the instruction word to be 16 bits:

Bit Position	15	14-8	7-0
Content	Type	Data	OpCode

Type Definition

Type	***Description***	***Data Field***
0	Immediate	Immediate value
1	Data	Number of 32-bit data

OpenGL Routine to Instruction Specification

OpenGL Routine	***Type***	***Data Field***	***Opcode***
Primitives			
glBegin (GLenum mode)	0	glBegin table	0000 0000
glEnd (void)	0	X	0000 0001
glVertex (TYPE x, TYPE y, TYPE z, TYPE w)	1	4	1000 0000
glColor (TYPE red, TYPE green, TYPE blue, TYPE alpha)	1	4	0100 0000
Transformation			
glLoadIdentity (void)	0	X	0001 0000
glLoadMatrix (const TYPE *m)	1	16	0001 0001
glMatrixMode (GLenum mode)	0	glMatrixMode table	0001 0010
glMultMatrix (const TYPE *m)	1	16	0001 0011
glPopMatrix (void)	0	X	0001 0100
glPushMatrix (void)	0	X	0001 0101
glRotate (TYPE angle, TYPE x, TYPE y, TYPE z)	1	2 (sin angle, cos angle)	0001 1000
glScale (TYPE x, TYPE y, TYPE z)	1	3	0001 1001
glTranslate (TYPE x, TYPE y, TYPE z)	1	3	0001 1010
glViewport (GLint x, GLint y, GLsizei width, GLsizei height)	1	4	0001 1011

glBegin Table

Constants	*Immediate Value*
TRIANGLES	000000

glMatrixMode Table

Constants	*Immediate Value*
MODELVIEW	001
PROJECTION	010
TEXTURE	100

3.2 ISA Discussion

It is obvious that our ISA could be better designed to be more elegant and optimal. However most of our efforts were concentrated on designing and implementing the graphics pipeline, hence we chose to adopt a straightforward and usable ISA which is easily extensible as we progressively extended the functionality of our pipeline.

A 16-bit instruction word size is an excessive design for the number of operations that we need to support. For example, our current architecture could easily be based on an 8-bit ISA. There are several factors that contribute to this choice:

1. Extensibility
   The OpenGL 2.1 specification defines up to 300+ routines. In addition there are a few libraries that our pipeline may want to support such as GLUT and the NV extensions.

2. Fetch unit
   The Instruction BRAM is addressed in 32-bit granularity. This is because we are fetching 32-bit fixed point data most of the time. Therefore having smaller word size will force us to put more effort in the OpenGL library to generate optimized VLIW to avoid wastage due to instruction misalignment.

3.3 Fixed Point

Our chip uses the following 32-bit fixed point format

Bit	Description
31	Sign
30-11	Pre
10-0	Post

Maximum integer is 2^20-1
Minimum fraction is 2^-11

We implemented add, multiple, divide and single-precision conversion for our fixed point format. The following are the specifications for each of the operations.

Add:
Asynchronous addition
LUTs utilization: ~1%

Multiply
Asynchronous multiply
LUTs utilization: ~1%

Divide
8-stage fully pipelined
LUTs utilization: ~11%

Convert
Asynchronous conversion from single-precision floating pint
LUTs utilization: ~1%

Optimization
Optimizing our floating point operations is very important because the performance of our graphics pipeline depends heavily on our fixed point functional units. We focused on improving the divider since it is both our largest component and our critical path. One tradeoff that was made is to reduce the precision of our divide in order to decrease resource utilization. We also had to pipeline the divider in order to meet our self-imposed 25MHz timing constraint.

4) FETCH UNIT

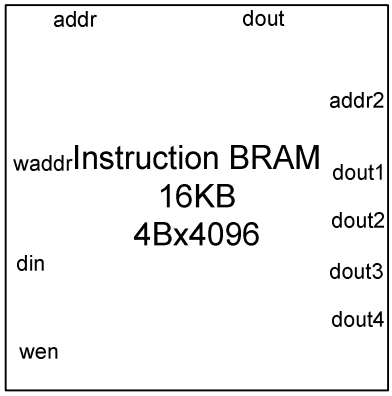
Figure 2 shows our fetch unit as part of our whole design. The fetch unit comprise of a server running on the PowerPC core, a 5-port read and 1-port write Instruction BRAM and the first two stages in the graphics pipeline.

4.1 Server on PowerPC core

This component is based on the Ethernet MAC OneWire sample from http://www.xilinx.com/univ/XUPV2P/Reference_Designs/edk_7_1_builds/xup_bsb_emac_onewire/xup_bsb_emac_onewire.zip which implements a web server on the PowerPC core.

We modified this sample to listen for instructions from the OpenGL library on the workstation. Instructions are written to the Instruction BRAM through the OCM bus as long as the pipeline does not send an Instruction Halt signal

4.2 Instruction BRAM



The Instruction BRAM is 32-bit width, 4096 entries Block RAM. We choose to have 5 read ports and 1 write port. This BRAM simultaneously serves the server, stage 1 and stage 2 of the pipeline. The server writes 32-bits into din at address waddr in every cycle. The port addr is used by stage 1 and addr2 by the second stage of the pipeline. Four 32-bit fixed point numbers can be read from addr2 using ports dout1-4.

Stage 1

Register Name	***Width***	***Description***
Program Counter	12	Current instruction address
Memory Bound Register	12	Address of the last instruction in the BRAM
Instruction Halt Register	1	Connected to the OCM Bus to notify the Server on the Power PC that the Instruction BRAM is full

The first stage of the pipeline fetches a 32-bit word using the address in PC. The top 16 bits stores the current instruction. Depending on the optimization done in the OpenGL library, the bottom 16 bits may be junk or could be another instruction. If it is another instruction the PC is stalled and the next instruction is buffered to be process in the next cycle.

The decode module is use to set control signals and update registers based on the current instruction.

Stage 2

Register Name	***Width***	***Description***
Data Address Register	12	Contains PC+1. This is use to fetch four 32-bit fixed point numbers from the instruction BRAM
Data Count Register	7	Contains the Data field of a Data instruction Indicate how many values fetched at address DAR are valid Used to increment the PC for Data instruction type
Model View Stack Pointer	8	Points to the top row of the first matrix in a 32 matrices stack
Projection Stack Pointer	4	Points to the top row of the first matrix in a 2 matrices stack
Matrix Mode Register	3	Select the active matrix stack
Compute Select Register	4	Select code for the shared fixed point array shared between the Matrix Stack Update and Transform pipeline

This stage retrieves 4 32-bit fixed point numbers for consumption of the subsequent pipeline stages.

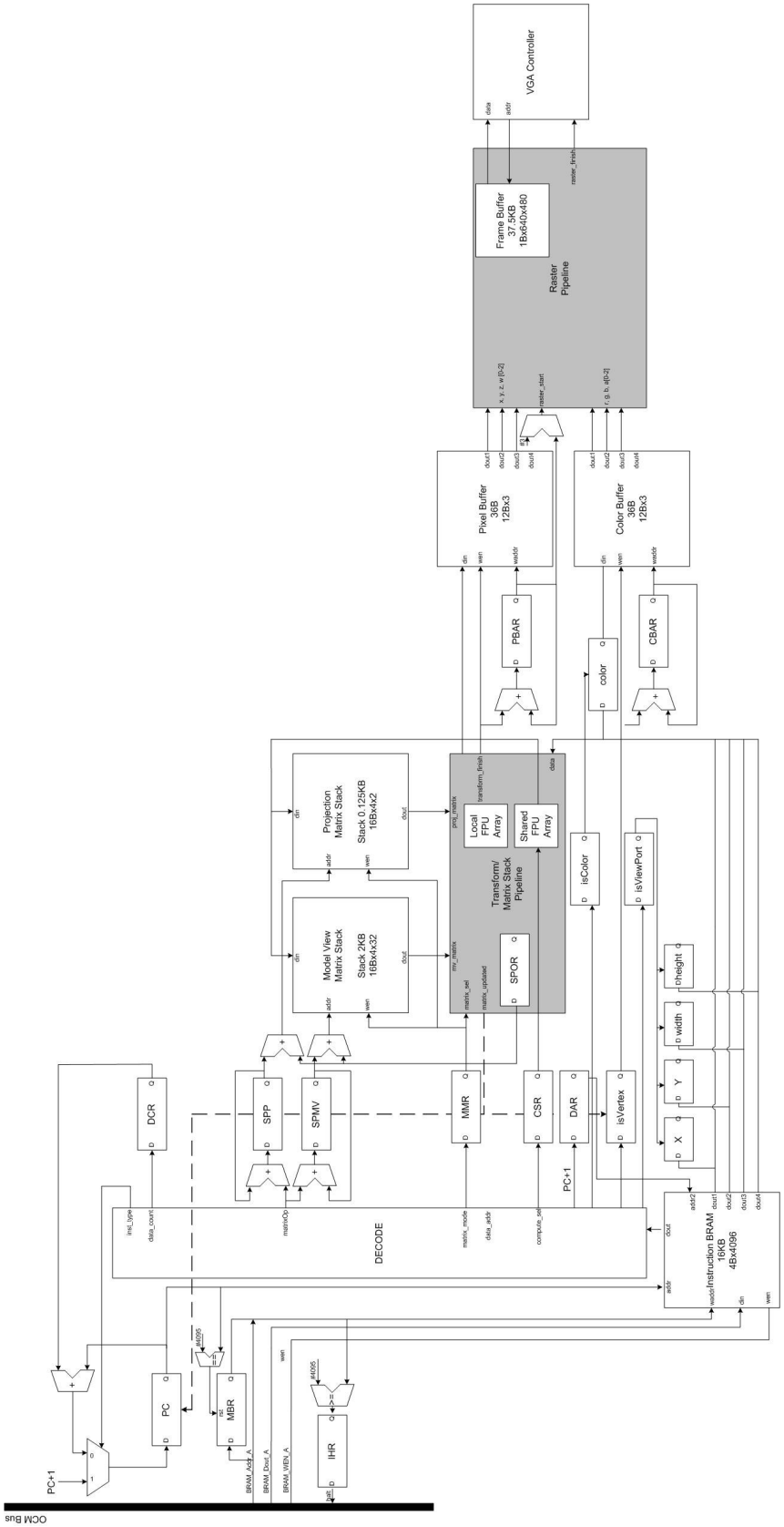


Figure 2: Fetch Unit

5) MATRICES & TRANSFORMATIONS

5.1 Matrix Stack Update

Our pipeline implements two matrix stacks (each storing multiple 4x4 matrices) namely the Model View and Projection matrix using BRAMs. These stack pointers are initialized to a default identity matrix at the bottom of those stacks. Since the transformation pipeline depends on the current transformation matrix (determined by glMatrixMode), the matrix stack update pipeline will stall all pipeline stages before itself in order to starve the transformation pipeline.

Since the matrix stack update pipeline is designed to update one row of a particular transformation matrix at a time, the pipeline is guaranteed to stall 4 cycles for every matrix update operation. This could be overcome if we can instantiate enough arithmetic units to compute matrix multiplications in one cycle. However, this is a tradeoff issue as we have limited resources on the FPGA.

Matrix stack updates are performed using the following OpenGL routines:

These routines involve writing a new 4x4 matrix on to the current matrix stack:
void glLoadIdentity (void)
void glLoadMatrix (const TYPE *m)

This routine involves the decrement of the stack pointer:
void glPopMatrix (void)

This routine involves copying the top matrix on the current matrix stack to position stack pointer + 1 on the stack:
void glPushMatrix (void)

These routines involve multiplying the top matrix on the current matrix stack with a 4x4 matrix (generated based on the routine's arguments) and overwriting the top matrix with the result of the calculation:
void glMultMatrix (const TYPE *m)
void glRotate (TYPE angle, TYPE x, TYPE y, TYPE z)
void glScale (TYPE x, TYPE y, TYPE z)
void glTranslate (TYPE x, TYPE y, TYPE z)

Each matrix update takes 4 clock cycles, updating each row of a matrix in each cycle. This is done using the computation structure in Figure 3.

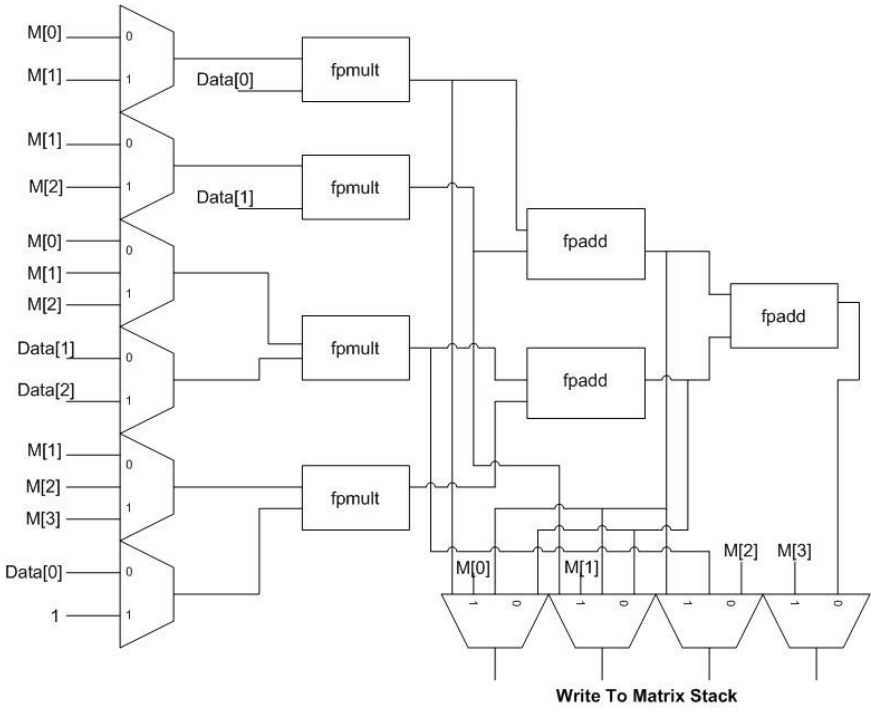


Figure 3: Row Matrix Computation

The inputs M are obtained from the matrix stack while inputs Data are new data from the instruction BRAM. Depending on the current OpenGL routine, the multiplexers will choose the appropriate inputs and outputs for the computation. The outputs of the computation are written onto the matrix stack.

5.2 Coordinate Transformation Pipeline

Coordinate transformations are performed when glVertex is processed. glVertex provides 4 arguments, i.e. the x, y, z, and w coordinates for each vertex. The transformations performed are shown in Figure 4.

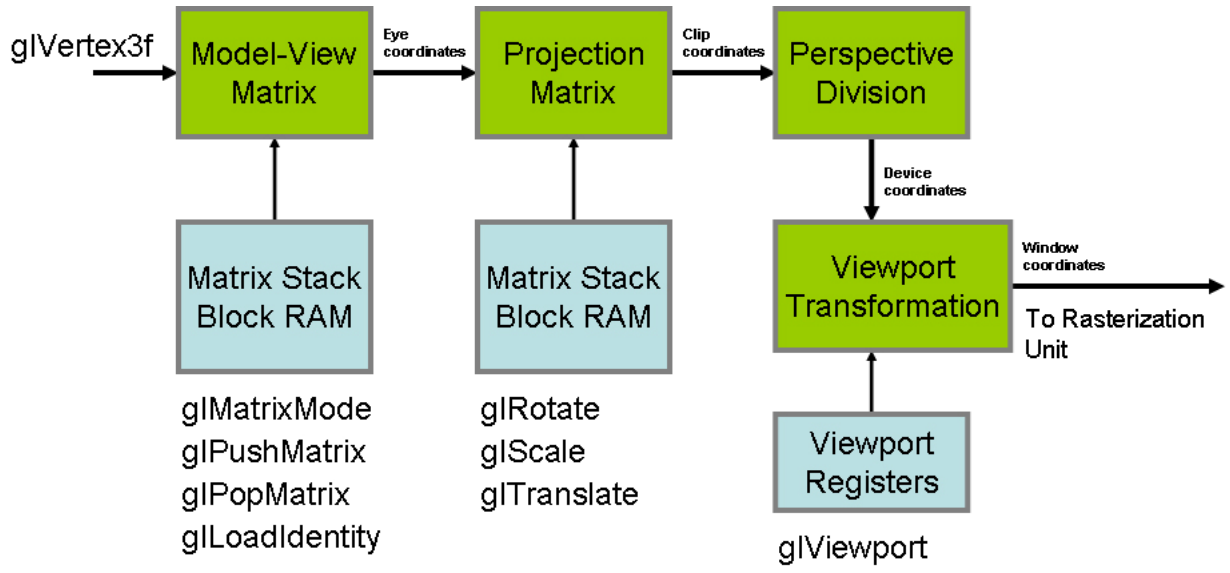


Figure 4: Coordinates Transformation

Since matrix updates and matrix transforms are performed on different clock cycles, we are able to share the computation units in matrix updates described in section 5.1 to perform the operations in the Model-View Matrix block in Figure 3.

The detailed block diagram for the coordinate transformation pipeline is shown in Figure 5.

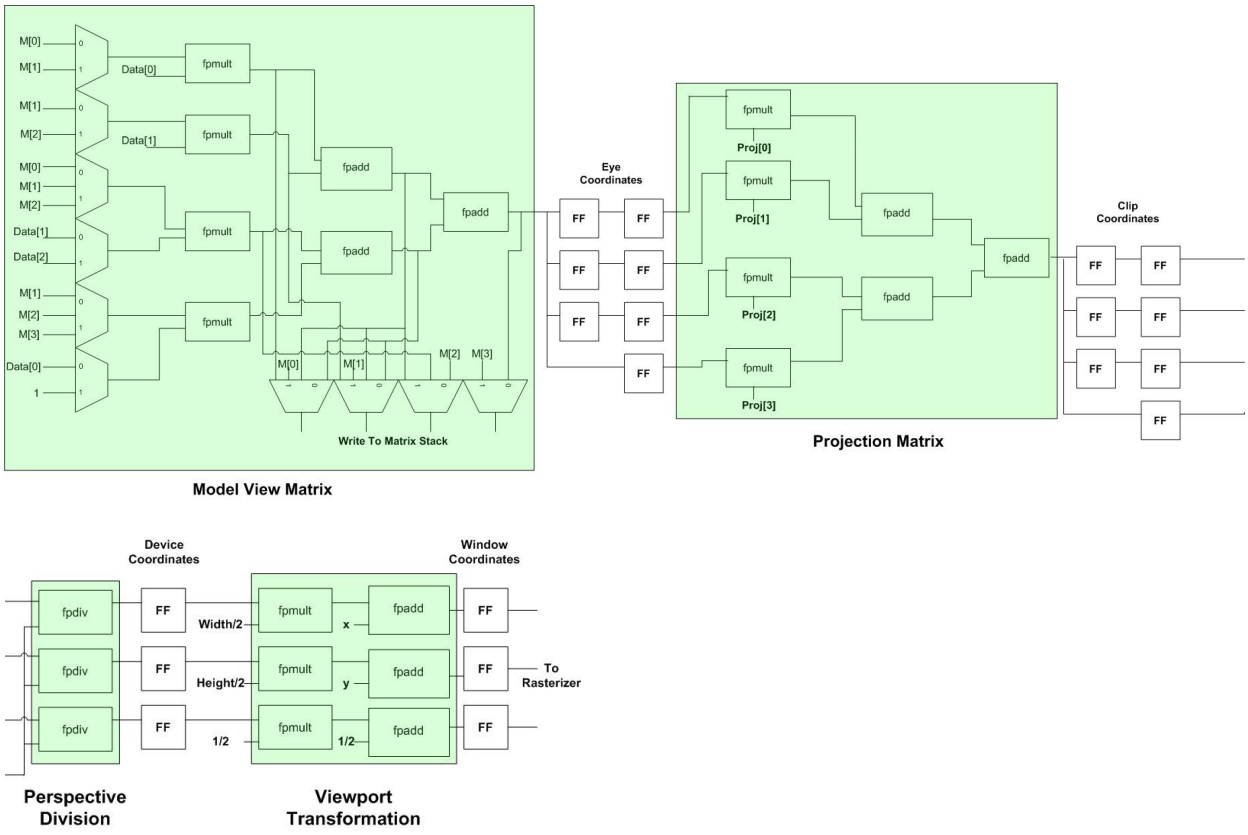


Figure 5: Coordinate Transformation Block Diagram

6) RASTERIZATION

Since all polygons can be broken down into triangles, our graphics pipeline needs to support triangle rasterization. Triangle is the most basic polygon that will enable us to render any graphics. We often want to draw a 2D triangle with three 2D points. Therefore, bounding box, scanline and edge walking is used to rasterize a triangle. In addition, the algorithm should also interpolate color or other properties from values at the vertices. The color information is determined using the *barycentric* coordinates.

4 main components for rasterizing a triangle:

- Bounding box
- Edge equation
- Scanline evaluation
- *Barycentric* coordinates

6.1 Algorithm

The algorithm (Figure 6) will perform a horizontal scanline within the bounding box (Figure 7) instead of the entire screen. This will increase the performance tremendously as we do not need to scan the entire frame (640x480). At each pixel, the algorithm will evaluate whether it is above the edge equation, on the edge equation or below the edge equation (refer to Figure 8 for a better visualization). The *barycentric* coordinates will normalize the number between +/- 0 and 1. If the current pixel satisfies all three equations (Figure 9), the algorithm will then draw the pixel. The color interpolation will use the *barycentric* coordinates: α, β and γ (refer to Figure 7 for the calculations). The algorithm will continue evaluating each pixel until y and x hits the max value of the bounding box.

```
xmin  = floor(xi)
xmax = ceiling(xi)
ymin = floor(yi)
ymax = ceiling(yi)

for y = ymin to ymax do
  for x=xmin to xmax do
    • = f12(x,y)/f12(x0,y0)
    • = f20(x,y)/f20(x1,y1)
    • = f01(x,y)/f01(x2,y2)
    if(• > 0 and • > 0 and • > 0)
       c = •c0 + •c1 + •c2
       drawpixel(x,y) with color c
```

Figure 6: Simplified Pseudo code for triangle rasterization

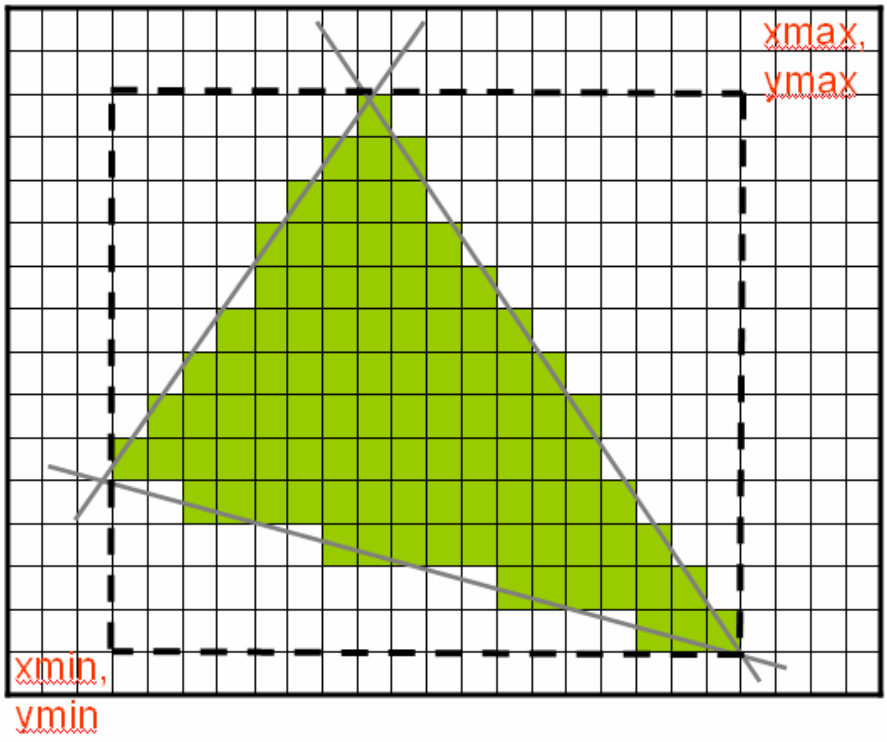


Figure 7: Bounding box

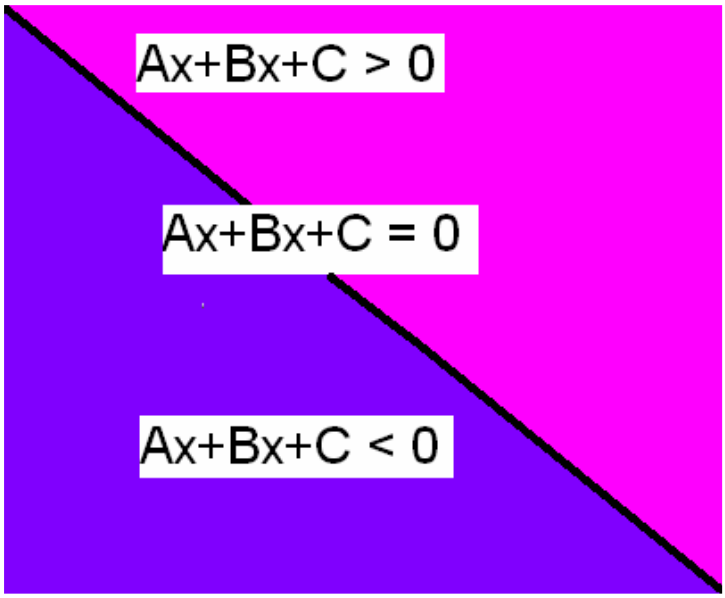


Figure 8: Edge Equation

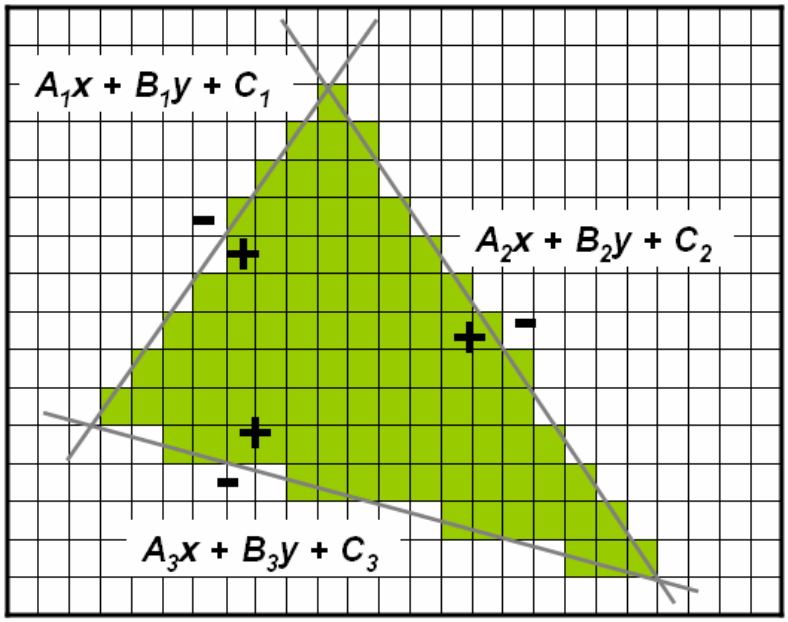


Figure 9: Pixel Evaluation

The algorithm will loop through each scanline and evaluate the pixel in order to determine whether it's positive, negative or on the line. If the pixel evaluates to positive for all three line equations, the code will perform color interpolation calculations. From these calculations, it will then determine the intensity for each color for the individual color buffers.

6.2 Color Interpolation

Barycentric coordinates are used in order to determine the color of each pixel. The *Barycentric* coordinates will be multiplied by c0, c1, and c2, which are obtained from the OpenGL instruction glColor(). c0 is the color from vertex 0, c1 from vertex 1, and c2 from vertex 2.

Below is the equation to determine the color of each pixel:

$$c = c0\alpha + c1\beta + c2\gamma$$

After evaluating c, it will then compare with a threshold number to determine whether the color will be drawn on the pixel or not. Since we're using 1 bit for each color (r, g, b) we will only compare with one threshold number, either on or off. There are no intensity involve in each color as there is only 1 bit representation. We limit the number of bits for color representation as there are limited BRAMs for the entire design. Figure 10 is the result of using color interpolation with 8 bits representation per color. We would obtain the same results if we use 8 bits per color as well.

In addition, we also implemented Gouraud smooth shading with the *Barycentric* coordinates Gourand shading gives a better effect than flat shading. *Barycentric* coordinates is very useful for other implementation as well. If we have the time an opportunity to work on other parts of the pipeline, we could use these coordinates for texturing and lighting.

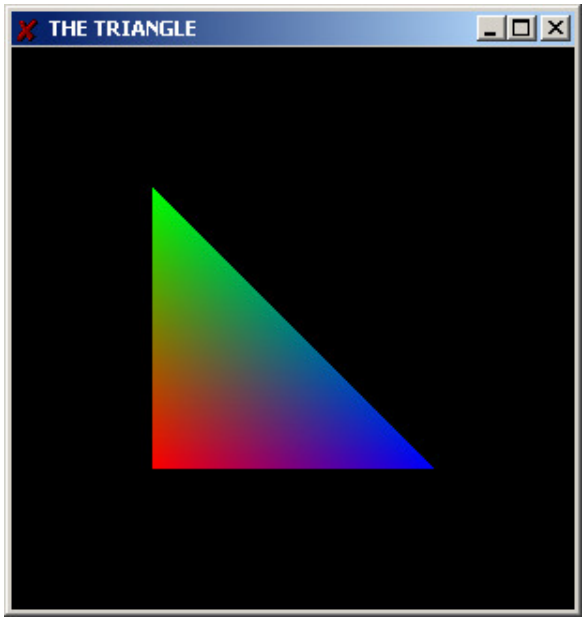


Figure 10: Color Interpolation

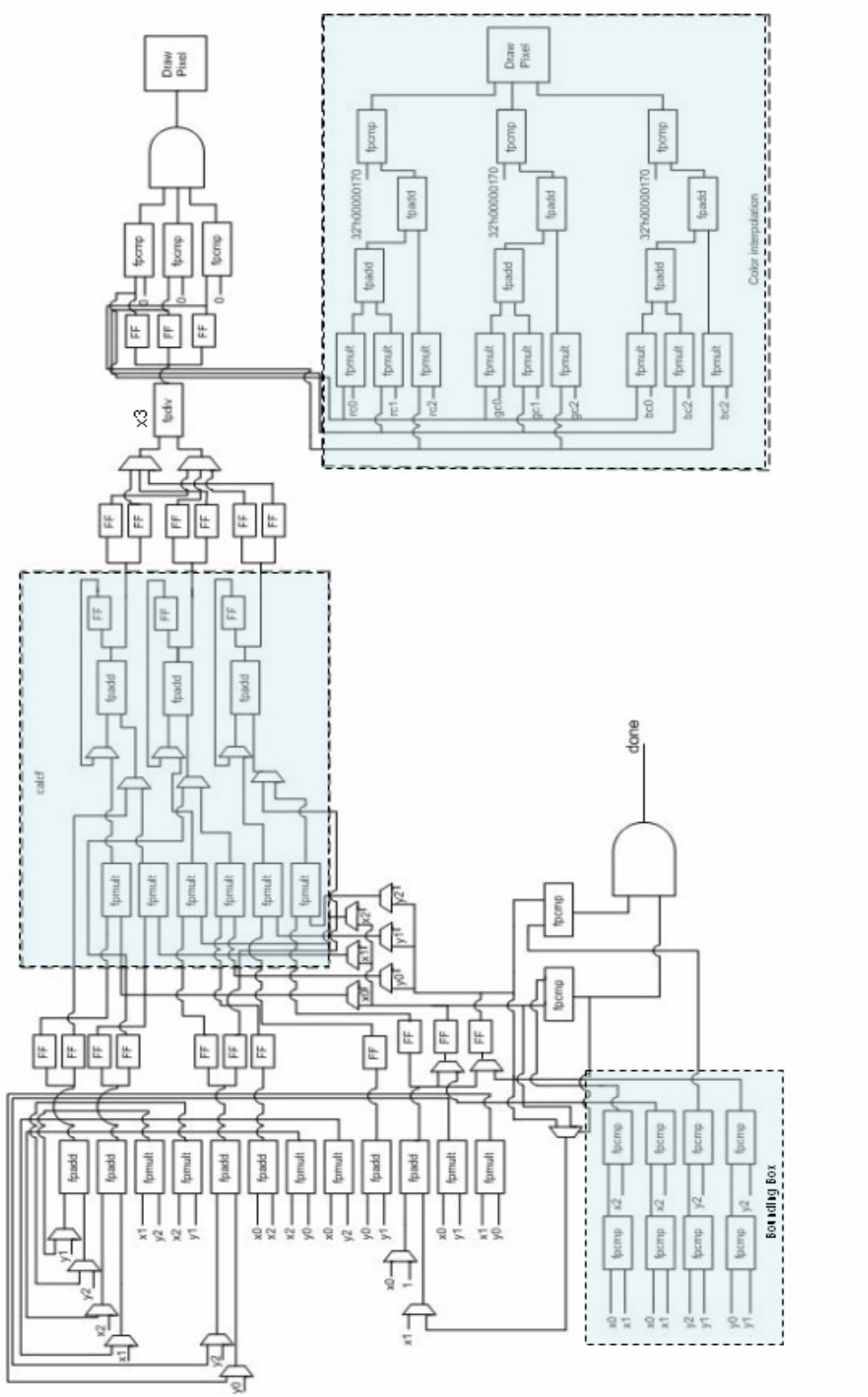


Figure 11: Block Diagram for triangle rasterization

The block diagram in Figure 11 implements the pseudo code (Figure 6) in Verilog. We did our best to reuse as the resources in order to have a smaller design and also to make sure that our design is able to fit onto Xilinx board. The major part of the block diagram is computing the calculations in order to determine whether the pixel should be drawn or not. The second major part of the rasterization pipeline is computing the *barycentric* coordinates for the color interpolation. There are also 2 loops needed for the scanline.

Below is the simulated output without color information:

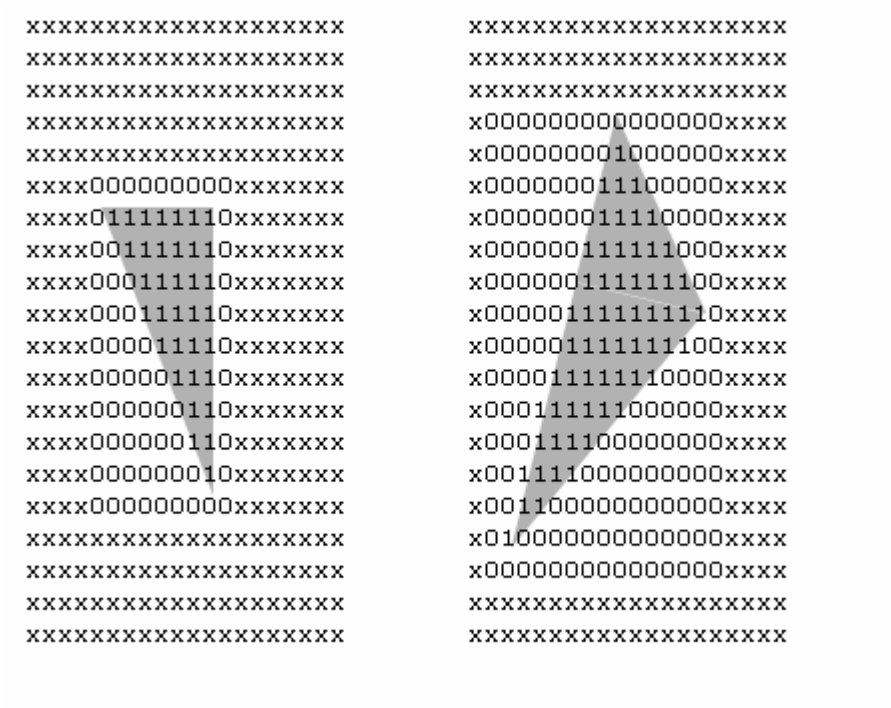


Figure 12: Simulation output

7) FRAME BUFFER

The frame buffer provided by Xilinx was mainly meant to be manipulated in a C application program that is run on the PowerPC. This meant that all image data that is to be displayed will be written into the frame buffer in C.

So the way the Xilinx TFT Controller worked, is with 3 major components,

- The actual main frame buffer memory
- An intermediate BRAM memory, big enough to contain a line (640 pixels)
- The VGA controller itself for generating the horizontal_sync and vertical_sync signals for transmitting RGB bit data onto the screen

The way it functions,

- Your C code would be the one writing into the frame buffer memory anytime it pleases. So whenever you wanted to display an image onto the screen, you would write to that frame buffer memory.
- The BRAM, which is large enough to hold a frame line (640 pixels), will retrieve a line worth of pixel data from the main memory.
- So the VGA controller is constantly reading off the BRAM as it traces across the screen, meaning it displays that line that is contained by the BRAM.

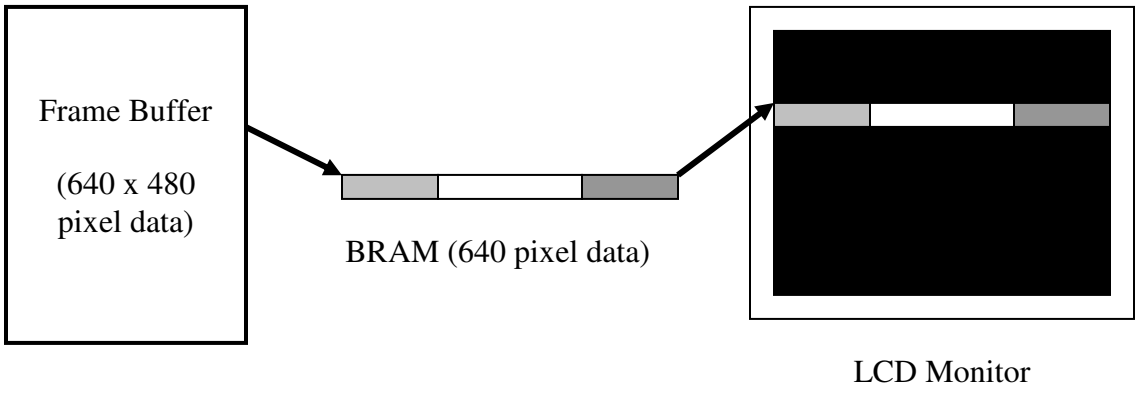


Figure 13: Xilinx TFT Controller IP

- So after that one line is done and completely displayed onto the screen, the BRAM will extract the **next** 640 pixel line from the frame buffer memory, and the VGA controller emits the horizontal_sync signal so it shifts one line down and displays the next 640 pixel line on the monitor.

Originally, it was our plan to have a complete Verilog driven frame buffer. The advantage would be that we would have complete control of the controller, without having to resort to using C code.

Additionally, direct access to the frame buffer would eliminate the need for data to go through the PLB bus which could take many cycles, thus speeding up the frame update and ultimately the frame rate.

It has been defined that the standard method for outputting VGA signals is shown in Figure 14:

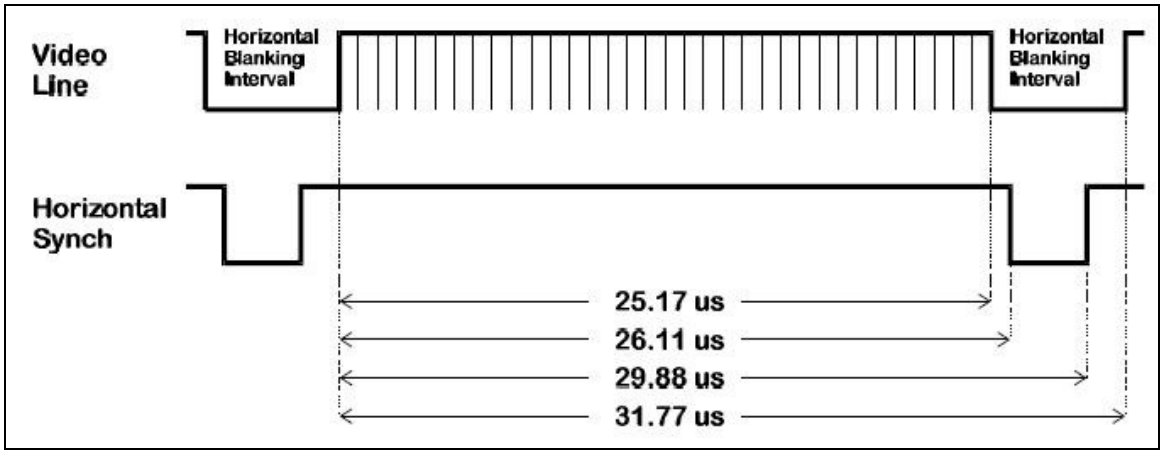


Figure 14: 640 x 480 @ 60Hz Non-interlaced mode timing for Horizontal sync

As can be seen in the Figure 13, there is a blanking interval during which the video signals must be low. The time for the blanking interval is 6.60 µs, which is 660 cycles at 100 MHz.

Having a Verilog driven controller is still advantageous to use in that it enables us to write what we want, when we want. If we were to use C for writing to the memory, we would have to synchronize the outputs of our rasterizer to the C code, so that they would write at the appropriate time.

If we were to approach it this way, the process would be as follows:

- Rasterizer will produce the x and y coordinate to color.
- From Verilog, the information will go to the C code which will then write it into the frame buffer memory through the PLB bus.
- From the frame buffer, the data will be read by the VGA controller through the PLB bus again.
- Then only finally the VGA controller can generate the signal onto the screen.

Such an approach was deemed somewhat clumsy and unnecessarily long winded. The data has to travel from hardware into software then into hardware again. It would be much simpler and cleaner to integrate the frame buffer and VGA controller directly to the rasterizer and have everything on hardware.

However, Xilinx does not provide the Verilog code for the DRAM controller IP for our own use as code. It could only be implemented under system assembly and manipulated via C instructions. Thus we decided on the implementation of utilizing the BRAM as our frame buffer, at the cost of color information.

We also implemented double buffering in order to remove all drawing artifacts during the drawing process. Double buffering is accomplished simply by having 2 separate frame buffers, one for reading, and one for writing. So, while one buffer is being read, the other buffer is being written onto. And after the rasterizer has finished for the current frame, the roles of the buffers are then swapped, the one that has finished being written will be read, while the other one will be cleared and written into.

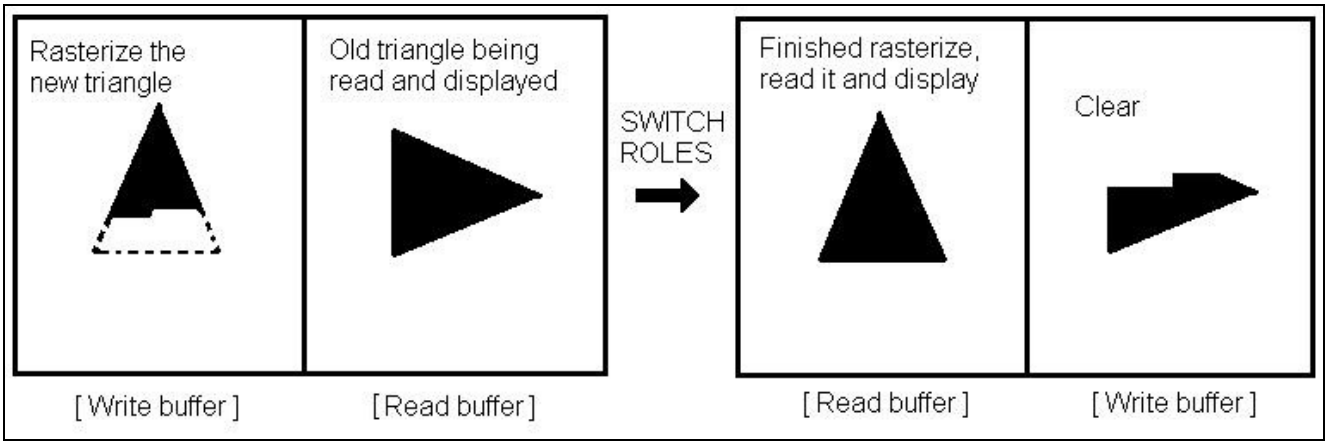


Figure 15a: Double buffering

Since double buffering introduced the need for double the amount of data storage, we reduced the resolution to 320 x 240. Instead of rewriting the VGA controller to sync for a 320x240 resolution, we just display each pixel on the 320x240 buffer twice on the display.
Color information was now represented by 1 bit per RGB color, providing a total of 8 colors on screen.

So, we utilized the BRAM to produce a 320x240 resolution, 3-bit color image.

Integration between the Rasterizer and the VGA Controller requires careful synchronization to ensure that the Rasterizer transmits the right data to the correct frame buffer. The double buffering requires that only one frame buffer be open for writing at any given time, while the other would be read from.

In order to enable the drawing of more than one triangle at any given time, the rasterizer must be allowed to write all desired triangles into the frame buffer before switching buffers for reading and writing. So the rasterizer will transmit a done signal to indicate to the VGA to switch buffers and read the newly finished rasterized image.

Then, for movement, the old frame buffer must be cleared before writing new triangles otherwise it would result in image 'trails'. The 'trails' would simply be the old triangle on the buffer that was there because of the prior write session.

Also, the rasterizer must also keep track of the VGA controller's actions. It cannot be allowed to begin raster unless the buffer has completed clearing the screen, otherwise, the clearing may clear parts of the newly rastered triangle. So a signal is sent to rasterizer to indicate completion of clear.

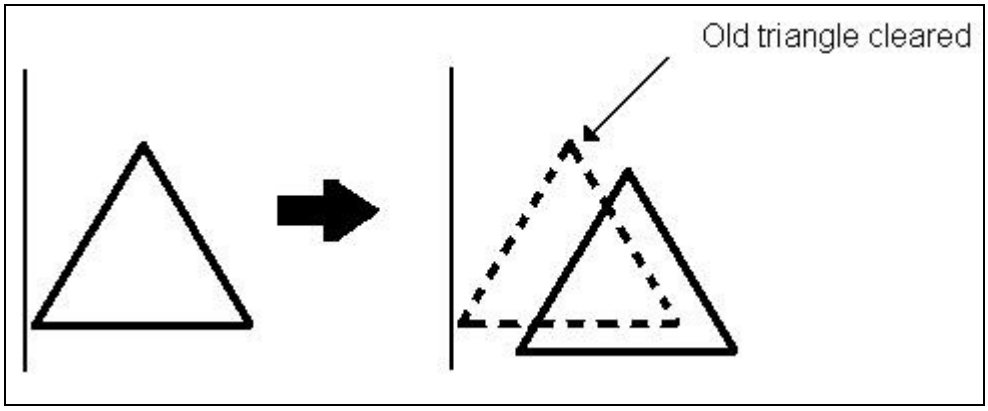


Figure 15b: Clearing to give the impression of triangle movement

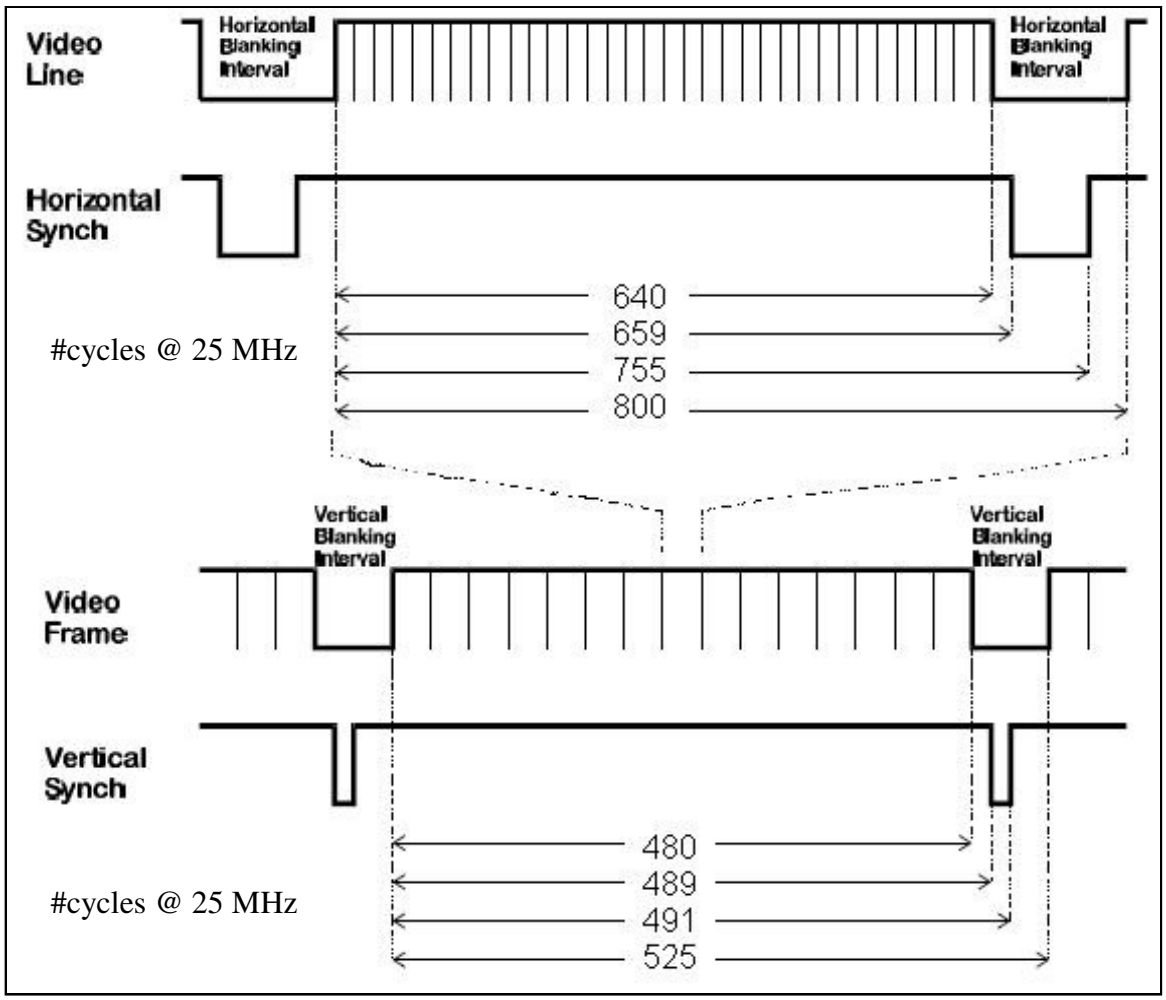


Figure 16: VGA Timing

8) DESIGN APPROACH

8.1 Design Partitioning

We partitioned our design into 4 main parts namely Fetch & Decode, Coordinate Transformation, Rasterization, and Frame Buffer. Each part was implemented with limited dependency on other parts of the design. To enable better integration, we specified the various requirements of each of the 4 parts before dividing up for individual development, such as number of clock cycle, whether input-output of a module are latched, control signals, and port requirements. Once each part was completed, we combined Fetch & Decode with Coordinate Transformation as the front-end, and Rasterization with Frame Buffer as the back-end of our design. The front-end and back-end of the design were then integrated to complete the pipeline.

We also identified modules which can be used across parts of our design. These included fixed point units and Block RAMs. We developed these modules such that they could be customized (port width, memory depth, and number of pipeline stages) to serve the needs of various parts of our design while maintaining their inherent functionality. This enabled us to eliminate duplicated development efforts.

8.2 Design Methodology

Our design methodology roughly followed the spiral model in software development. The methodology is shown here in Figure 17.

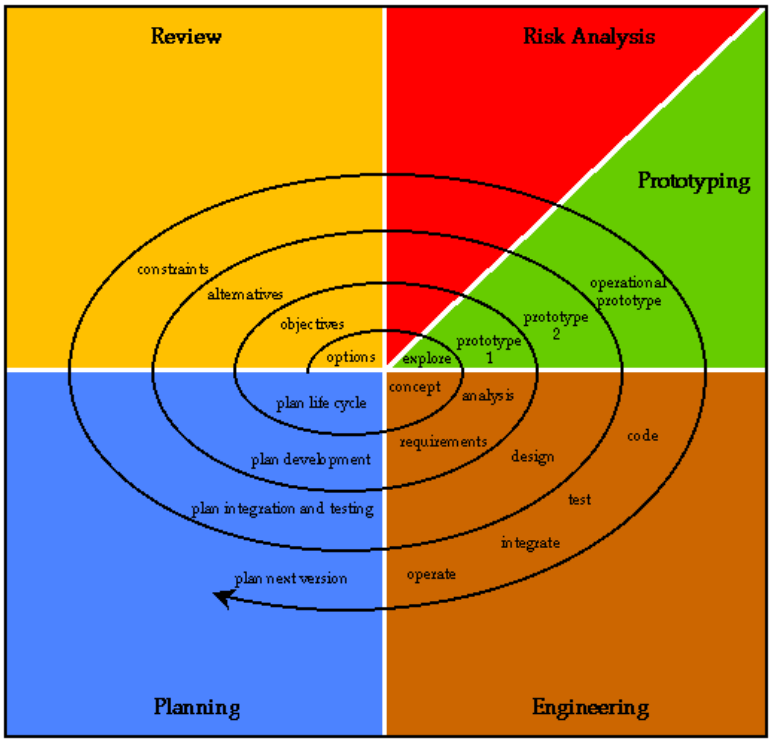


Figure 17: Spiral Model
http://www.csse.monash.edu.au/~jonmc/CSE2305/Topics/07.13.SWEng1/html/text.html

The spiral model emphasizes the iterative development process whereby requirements are refined as the development process go along. It is an attempt to combine the advantages of top-down and bottom-up development methodologies. This model was chosen because of its several advantages which fits our project requirement:

Iterative refinement of requirements
Given that we were beginners in computer graphics, it made sense that questions popped up throughout the semester-long development process. By allowing the iterative refinement process, we were able to redefine requirements quickly as we learned more about OpenGL and designed new prototypes. We were able to estimate more effectively what was doable and what was not as we went along. If we used other methodologies which focused on design requirement analysis during the planning stage, there is still a higher change that we ended up having to change our designs due to the limited level of knowledge that we started with.

Timing closure
Protracted design planning process is not feasible in this project, due to the limited amount of time that we have in a semester. We believe that issues are bound to arise during the development process and it was important that we are able to discover them earlier rather than later. Therefore, we did not spend too much time on the planning stage in order for work to begin earlier.

8.3 Testing & Verification Methodology

We tested partitions of our design separately before integrating them for final testing. Given the scale of our design, it was easier to identify bugs by testing on smaller modules before combining them into big ones. Since we identified the inter-module requirements prior to the module development, we were able to come up with possible test vector quickly. We used ModelSim as our Verilog simulator for its convenient graphical display of all possible signals in our design.

The synthesis and place-and-route reports by Xilinx were useful in determining problems with our design which were not visible using simulation. Timing constraint violations were quickly identified and fixed, as the critical path is specified clearly in the reports.

We verified our final design output with those produced by OpenGL code in software. By using the same inputs and comparing the outputs produced by a software and hardware implementation, we were able to easily identify any discrepancies.

9) STATUS & FUTURE WORK

Our design is architecturally complete. Currently, the whole pipeline design is not able to fit on the board with 103% LUTs utilization. The Rasterization Unit and the Frame Buffer are synthesized and they work perfectly to produce the intended output on the VGA. The Fetch & Decode Unit and the Transformation Unit simulate correctly and does not have synthesis issues if synthesized independently from the back-end of the pipeline.

Moving forward, we need to optimize our design to reduce the amount of resources used. Instead of using BRAMs for the Frame Buffer, we should explore the possibility of using DRAMs on the board. This will enable us to display more color variation given the color interpolation capability of the Rasterization Unit.

It would be nice to add more stages into our pipeline, such as lighting and texturing. These features will produce a more realistic look. Since we already have the *Barycentric* coordinates, which are used in this stage, we could add more matrix manipulation for the computation. However, currently our design is over mapped on the board. Thus, we will have to optimize the current pipeline stages first before adding new stages.

10) LESSONS LEARNED

We believe that we would have made much more progress this semester if we had computer graphics background before we started this project. By knowing enough about computer graphics, we would have been able to carefully plan out our design without the risk of spending too much time on it and not getting the results that we wanted.

It would be much easier if we have some exposure on Xilinx board. We would not have to spend so much time reading up the manuals and tutorials. By knowing the constraints and limitation of the board would also help us designing our project. A better understanding on Xilinx's optimization tool is important to understand the tool's behavior.

We were very ambitious in the beginning of the semester. We thought we could implement the whole graphics pipeline and run a game, FooBillard with our graphics accelerator. Without any background in OpenGL and Computer Graphics, it was actually impossible to achieve in a limited time frame. Most of the time spent in the earlier part of the semester is to gain understanding on both OpenGL and Computer Graphics via the internet and books. We did not realize that we have to change our goals until design review 2. Therefore, it was a mistake being too ambitious in the beginning. If we realized earlier, we could have been more focus on the graphics pipeline instead of the game.

At first, we also implemented most of the graphics pipeline using floating point representation. We thought that floating point representation would give us more accuracy and we found some open source floating point unit online. Later, we discovered that the floating point unit could not synthesize on Xilinx board. Thus, we had changed the floating point unit, so that it would synthesize. However, the utilization of the floating point unit is too high. Then, we changed everything to fixed point notation instead. Although we'll loose a bit of precision, fixed point notation give us better utilization and performance in general. We would say that we wasted a lot of effort and time in deciding which representation is suitable for the notation. We had to redesign the whole pipeline when the basic arithmetic unit changes. In brief, we should have used fixed point notation earlier and avoid redundant implementation of the pipeline.

In addition, we also tried to implement everything on hardware. For instance, we avoided using the PLB bus and the DRAM for the framebuffer. Since most of the pipeline is in hardware, we figured that the vga controller should be in hardware as well since the cycles count from the BRAM is much less than accessing memory from DRAM. However, by imposing this decision we had added more constraints into our design. Since we have limited BRAM, we had to be really tedious in allocating the BRAM through out the whole pipeline. As a result, we could only afford to represent 1-bit color information for the color interpolation. Thus, our triangle does not look as smooth as it should. When we realized this issue, we were running out of time to change the direction of our design.

By choosing this project (graphics accelerator), we managed to learn a lot from it. Since we're implementing our design purely in hardware, we learned how to deal with integrating several blocks together, synchronizing the timing between different blocks, designing our own architecture for the fetch unit and fitting everything on the Xilinx board. Hardware is a lot harder than implementing Software at a higher-level. Furthermore, we are exposed to OpenGL and

Computer Graphics by choosing this project. Some of us would never have seriously gotten into the graphics world if not for this project.

In addition, we also learn how deal with design closure. The initial description of the project has to meet a growing list of design constraints and objectives. There are so many constraints that we need to take into consideration, such as timing constraints, placement and routing, utilization constraints, mapping and logic synthesis. We must be able to design around those constraints.

For those who are interested in embarking on the same project, we strongly advise that you think through your decision carefully before doing it if you do not have any computer graphics background and if time and manpower is a constraint. It is also important that you make careful decisions in the beginning of the project by considering all the options and alternatives that are available. It will be much harder to turn around during the later stages of the project.

11) INDIVIDUAL COMMENTS

Joomay Tan

The earlier stage of the project, we spent some time determining what kind of project that we'll work on through out the entire semester. There so many possibilities and options available. We spent about two weeks defining our project and finally we came up with implemented a graphics accelerator. From the onwards, I spend a lot of time digging into OpenGL and Computer Graphics. I had no OpenGL or Computer Graphics background at all. Thus, it took me some time to grasp the pipeline implementation and obtained a good understand for each stage of the pipeline that we are going to focus on.

I started the triangle rasterization pipeline approximately a week before design review 1 and completed the initial implementation in around 2 weeks. After the initial implementation, I did a lot of testing and verification to make sure that the Verilog is able to render all kinds of triangle. Throughout the entire project, I did a lot of modification for the triangle rasterization. Initially we implemented everything with floating point, but later we decided to use fixed point representation instead. We also changed some modules that triangle rasterization depends on. Thus, I had to modify the code in order to fit everything together and make sure it works accordingly.

After that, I assist Ken Yu with the integration between the rasterizer and vga framebuffer. There were some synchronization issues and we had to do a lot of debugging. We had to make sure that the output of the raster fits into the vga controller. We also had to implement double buffering for the animation. I also implemented color interpolation into the pipeline. The color interpolation is embedded inside triangle rasterizer. With this added feature, we had to change the framebuffer to support all three color components, red, green and blue. There was a lot of testing, debugging and synthesizing done along the way.

In general, this is a fun class as every team gets to design and implement their own system. We learned how to put a huge system together, which is not obtainable from other classes. We also experience how to work as a team in this class.

One machine for each group is not enough as we always have to take turns using the machine and the board. A lot of time is wasted since the synthesizing process takes quite long especially if the utilization of the board is high.

I think everyone in the lab has the responsibility to keep the lab clean. However, this did not happen. The lab is constantly dirty and smelly. Moreover, our lab bench is next to the trash bin and the awful stench is always there. In addition, there are not enough lab stools for everyone in the lab.

Ken Yu Lim

Initially, I was involved in designing the interface between the workstation and board. I had linux booted onto a laptop, and planned to utilize the SATA connection. However, this idea was later dropped as writing a device driver and interconnecting to the SATA interface was too cumbersome and time consuming.

For the project, I dealt with mainly the back end of the pipeline. I was to implement the VGA controller in hardware. It was difficult at first, as I was now aware of the definitions of the required VGA signals for frame buffers in general. I learned a lot about getting data to sync correctly with the VGA signals in order to accurately relay pixel data from a frame buffer onto a graphics display. A lot of time was wasted trying to understand the intricate details about the VGA sync signals and looking at the Xilinx TFT controller IP which was provided in Verilog, fortunately.

We started with a 640x480 resolutions. I had all the sync modules and the memory all arranged to perform the tasks at that level. However later on, it was discovered that we would be utilizing the BRAMs and due to the limited availability, we had to drop the resolution. Hence I would have had to redo all the sync signals. So instead, I decided to reuse the same sync for 640x480 @ 60Hz, but to read each pixel twice onto the screen. Additionally, I also had to implement the 3-bit color information for the display.

In addition, I also was involved in the integration of the VGA controller to the rasterizer, ensuring that the outputs of the rasterizer would correctly write into the frame buffer. I also implemented double buffering for the frame buffer, so there was also the synchronization of the rasterizer to the appropriate frame buffer during the write, or rasterize period.

In general I learned a great deal about graphics processing and the general graphics pipeline, and the details of its workings and mechanics. I learned about the architecture, and how it all comes together. I also learned about some rasterization techniques and optimizations.

Overall, it was a very good experience. I learnt the hard way about designing with constraints, and about design closure, trying to ensure that the design will ultimately fit on the board and meet timing constraints.

It would have been more beneficial if there were clearer or more extensive labs on how to manipulate the display other than the C “setPixel” code method. Also, it would be good if there were Verilog code for the Xilinx DRAM Memory controller IPs. There are no very clear instructions about how to implement your own design modules in the beginning. There should be some explanations about the .mhs and .ucf files and how they come together.

Kenneth Eng

In this project, I was responsible in the following tasks:

Computer Graphics Knowledge
Since none of the team members have computer graphics background, I was assigned to gain an in-depth understanding of concepts and algorithms in computer graphics to get the team started on the project. By reading up books on computer graphics and researching online, I collected the necessary information to facilitate the pipeline design. The amount of time spent on this was slightly more than a week. However, learning and relearning of computer graphics was done throughout the duration of the project in order to refine the requirements of the pipeline.

Coordinate Transformation Unit
I was responsible to implement to the coordinate transformation module. It mainly involved the understanding of matrix multiplications and discovering methods to optimize its performance. By studying the mathematics of matrix transforms, I was able to perform 4x4 x 4x4 matrix multiplication using the same amount of hardware as a 4x4 x 4x1 matrix multiplication. By doing so, we saved some precious amount of resources on the board. The coordinate transformation unit also involves the implementation of the modelview and projection stacks using Block RAMs. I have to change my design during the later stage of the project in order to solve timing constraint problems during synthesis.

Synchronization between Decode and Transformation Unit
I worked closely with Teck Hua to synchronize the OpenGL instructions and data with the transformation pipeline's operations. Since most of the OpenGL routines directly affect the transformation operations, this process took almost 3 weeks before we are able to solve all the discovered synchronization problems between the two units.

The difference between what our team is doing compared to other team is that we are not able to see "tangible" results every time we got something to work. Most of the time, our level of self-satisfaction was limited to seeing correct signal behaviors on ModelSim. This had somewhat limited the level of enjoyment of the project. We were constantly living under the impression that we were not producing results even though we were churning out parts which contributed to the final pipeline. There were numerous times when we even doubted our decision in choosing the graphics accelerator project, but we pulled through nevertheless.

The labs could have been more instructive by providing more challenges and being better organized before the course started. More money should be spent on getting more lab benches with boards and other equipments. Having four persons working on a small lab bench is very inconvenient, so we tend to use remote desktop from other computer cluster during most of the times.

Teck Hua Lee

Fixed Point Library
We made the decision to switch from single-precision floating point to a custom fixed-point format around mid-way through the project. This is because we were using a freeware FPU from opencores.org, which is difficult to tweak to meet our resource, and timing constrains. I was responsible to write the fixed point units used in this project. As mentioned in the Fixed Point section, these are the crucial components that determine the size and performance of our GPU. There is very little room for error because every change here will require modification to the transform and raster pipeline. The first version of the fixed point library resulted in a 7MHz GPU. After rewriting the divider several times, we managed to crank up the speed to 60Mhz by the end of the project.

OpenGL Library client and PowerPC server
This is the interface between an OpenGL application and our GPU. The Ethernet plumbing on the client side is from the 15-213 Proxy Lab while the server code on the PowerPC is modified from the sample OneWire Ethernet project on the Xilinx website. I collaborated with the PPU team to iron out some performance issues with the Ethernet.

Fetch and Decode Unit
I did some research to see if a graphic processor would have implemented these stages differently from a general-purpose processor. In the end I modeled it after the typical fetch and decode stages in 18-447 RISC pipeline. I worked closely with Kenneth to ensure that the control signals asserted by the decode unit matches his transform pipeline specification. A key component is the Instruction BRAM which has 5 read ports and 1 write port. This allows the following operations to be done in the same cycle
1. write from the Server on the Power PC
2. instruction fetch
3. retrieve up to 4 fixed point data in the decode stage

Overall system integration
The most difficult part of the entire project is to connect everything together. I wrote the bridge between the Transform and Raster pipeline. This glue logic mainly defines the Coordinate and Color buffers which feeds the rasterizer. Since transformation is relatively faster than rasterization, we have to consider that trade off between using more BRAMs or stalling the pipeline very frequently. In order to free up BRAMs, we made a compromise by reducing the resolution of the VGA to 320x240. As a result we were also able to do double buffering and support up to 3-bits of color.

Comments
We are frequently being compared to a similiar project at UNC. However I think we faced a more challenging path since the UNC team is actually made up of the entire class of a Computer Graphics Hardware course. Hence they have a proper structure that guided them along the way through relevant lectures throughout the semester. Nevertheless I still enjoyed this class because it allow us to trip and fall so many times (redesign, redesign and redesign) and eventually we have a very good understanding of the OpenGL pipeline.

Complaints

The scope for the project is very flexible. There is a lot of uncertainty in the beginning of this project. Needless to say Prof. Mai was very helpful in helping us define our project. There is a big learning curve in the area of computer graphics and GPU architecture. By the time we were up to speed, our other classes were also hitting the peak period and time became really scarce.

12) CITATION OF RESOURCES USED

Shirley, P., Ashikhmin, M., Gleicher, M., Marschner, S., Reinhard, E., Sung, K., Thompson, W., Willemsen, P. Fundamentals of Computer Graphics, Second Ed. A K Peter, Ltd. 2005.

OpenGL Architecture Review Board, Shreiner, D., Woo, M., Neider, J., Davis, T. OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Version 2 (5th Edition). Addison-Wesley Professional. 2005.

Astle, D., Hawkins, K. Beginning OpenGL Game Programming (Game Development Series). Course Technology PTR. 2004.

OpenGL & OpenGL Utility Specifications. OpenGL Architecture Review Board. 15 Oct. 2006. <http://www.opengl.org/documentation/specs/>.

CS448A: Real-Time Graphics Architectures. Akeley, K., Hanrahan, P. 15 Oct. 2006. <http://graphics.stanford.edu/courses/cs448a-01-fall/>.

OpenGL Man Pages. 15 Oct. 2006. <http://www.cs.rutgers.edu/~decarlo/428/glman.html>.