18-545: OpenGL Graphics Acelerator

Andrew J. Lau, Alan X. Zhu, and Nathan L. Wan
{ajlau, axz, nlw}@andrew.cmu.edu
Carnegie Mellon University

December 8, 2010

Contents

List of Figures

List of Tables

1 Introduction

This report discusses an OpenGL graphics accelerator implemented on a Xilinx XUPV5-LX110T FPGA during the Fall 2010 iteration of the Advanced Digital Design capstone course at Carnegie Mellon University. We cover the overall system design, implementation details, and advice for adding further functionality to the system in the future.

2 System Overview

The graphics accelerator can be broken down into five major components:

1. Perl Parser
2. Instruction Assembler
3. Instruction Cache
4. Coordinate Transformation Pipeline
5. Rasterization Unit
6. Framebuffer/Display

¡insert system diagram here¿

3 System Specification

3.1 Hardware

- Microblaze processor
- Floating point units
- Coordinate Transformation Pipeline
- Rasterization Unit
- Instruction Cache
- FIFOs
- Framebuffer Controller

3.2 Software

- Perl Parser
- Instruction Assembler (running on Microblaze)

3.3 Development Software

- Xilinx ISE Design Suite 12.2

  READ: OS and Libraries Document Collection and EDK Concept, Tools, and Techniques

- Git

- Windows XP

- Ubuntu

  Initially, wanted machine to be primary machine for development. Unfortunately, the usb drivers for the JTAG cable never came into fruition, so board programming was relegated to the Windows machine. Much of the simulation and core development still happened on the Ubuntu box. Also held a git repository that was later moved to github. It was used as an easy way to transfer files between group members because Windows networked file system is unusable.

4 OpenGL

4.1 Rationale

We chose to implement an OpenGL pipeline because it is a well known industry standard for 2D/3D graphics. In addition, the pipeline design itself is well documented with numerous existing implementations in both hardware and software.

4.2 Supported Functions

4.3 Interface to Pipeline

4.3.1 Perl Parser

In order for the pipeline to accept graphical language calls, a Perl script was written that takes human readable trace of OpenGL calls and outputs a hex representation of the program.

The trace is generated from simple hand-generated C programs. Conditionals, loops, and variables are unrolled and substituted so that what remains is strictly OpenGL calls and data in the form of actual numeric values. These traces correspond to the calls that would be made by a C library such as GLSim.

The main function of the script is to parse each GL call, generate a 32-bit hex value that corresponds to the instruction code defined in the ISA, and output the hex value to a file. If there are argument parameters in the call, each parameter is translated into its hex representation and printed on subsequent lines in the file. The output is an OPNGG hex file (*.gg). Each line of the file is one hex value, either corresponding to an instruction call or one of several 32-bit arguments encoded in hex (single precision floating point , unsigned integer, etc.)

4.3.2 Instruction Assembler

The Microblaze processor is critical for programming the accelerator. To program the instruction cache, the Microblaze reads off a file containing the graphical program and copies in data. The file resides on the Compact Flash and its format guaranteed by the Perl Parser; the assembler does not check the format of the executable. The SDK's XilFATFS is a library that allows the Microblaze to read files on the Compact Flash. For the most part, it follows the C convention for file handling. The instruction BRAM has an interface identical to that of the Xilinx BRAM cores. This is so that a PLB to BRAM interface core, provided by Xilinx, can be used to allow the Microblaze to write directly into our module's BRAM.

5 Instruction Set Architecture

5.1 Specification

The ISA defines a 32-bit instruction word to align with the width of single precision floating point word. The *opcode* field is 8 bits wide, allowing support for up to 256 different routines. This is leaves plenty for room for extending the pipeline to support other some of the 300+ OpenGL calls, as only 17 slots are filled in the current state. The fetch unit addresses the instruction cache at a 32-bit granularity.

Bits	[31]	[30:8]	[7:0]
Content	Type	Data	Opcode

Table 1: Instruction word

Each supported function is translated by the Perl parser into the follow instruction words. The *data* field indicates to the fetch unit to expect a certain number of floats following the instruction word if the *type* field is set.

Function	Type	Data	Opcode
glBegin (void)	0	X	00000001
glEnd (void)	0	X	00000010
glVertex (float x, float y, float z)	1	3	00000011
glColor (float r, float g, float b)	1	3	00000100
glFlush (void)	0	X	00000101
glMatrixMode (enum mode)	0	imm	00010000
glMultMatrix (const float *m)	0	16	00010001
glLoadIdentity (void)	0	X	00010010
glLoadMatrix (const float *m)	1	16	00010011
glPushMatrix (void)	0	X	00010100
glPopMatrix (void)	0	X	00010101
glRotate (float angle, float x, float y, float z)	1	4	00010110
glScale (float x, float y, float z)	1	3	00010111
glTranslate (float x, float y, float z)	1	3	00011000
glViewport (int x, int y, int width, int height)	1	4	00011001
glFrustum (int left, int right, int bottom, int top, int near, int far)	1	6	00011010
glOrtho (int left, int right, int bottom, int top, int near, int far)	1	6	00011011

Table 2: OpenGL routine to opcode mappings

6 Coordinate Transformation

6.1 Matrix Operations

6.2 Perspective Division

6.3 Viewport Transformation

Viewport transformation scales and translates the normalized device coordinates to fit the rendering screen. The results (x_w, y_w, z_w) are passed to the rasterizer. The transformation is given by:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{w}{2} x_{ndc} + (x + \frac{w}{2}) \\ \frac{h}{2} y_{ndc} + (y + \frac{h}{2}) \\ \frac{f-n}{2} z_{ndc} + \frac{f+n}{2} \end{pmatrix}$$

Figure 1: Viewport Transformation

With some factoring, this is implemented using three floating point multipliers and five floating point adders.

7 Rasterization

7.1 Bounding Box

7.2 Horizontal Scanline

7.3 Color Interpolation

8 Framebuffer/DVI Controller

8.1 DVI Controller

The reference design for the XUPV5 contained a DVI controller that displays a frame buffer on DDR2 SDRAM. The core outputs a 640 by 480 video over the DVI port of the board. The core reads a line of the SDRAM frame buffer over the PLB bus and stores it locally in BRAM. Since the frame buffer is any 2MB region in the PLB address space, the frame buffer does not actually have to reside in SDRAM. Any mapped address can be used as a frame buffer, which demonstrates some of the flexibility of the PLB. The DVI controller has control registers mapped on the DCR bus interface. In order for other elements in the system to configure the controller, there is a PLB/DCR bridge what maps the DCR space to a part of the system PLB address space.

8.2 PLB IPIF

Using the PLB actually offered great flexibility in the interconnect of cores. Although it restricted the project in some ways, requiring the use of particular

8.3 DMA

9 Major Design Decisions

9.1 Floating Point vs. Fixed Point

9.2 Synchronization

9.3 External Cores

It soon became apparent we could not do everything. The decision to conform to the Xilinx tool chain stemmed partially from the convenience of provided Xilinx cores and generated cores from CoreGEN. The framework provided by the Xilinx EDK had some advantages. It has a way to run C code a soft processor, the Microblaze. It provides the interconnecting bus structure, PLB. The reference design also contains the Multi-Port Memory Controller (MPMC) which provides an accessible interface for the DD2 SDRAM.

9.3.1 CoreGEN

CoreGEN allows us to generate different types of modules. It is a tool provided by Xilinx included in their design suite and provides blackbox *.v and *.ngc files for usage in your projects. They provide a variety different, commonly used cores, that is, if it is not a custom type of hardware, you can probably find it in CoreGEN. Most of the modules in the reference design actually comes from CoreGEN, including the versatile MPMC. The floating point units used throughout our design were generated to specific parameters. It was useful to specify usage of DSP48E slices instead of logic slices. CoreGEN also allowed us to specify latency, which became extremely important. The FIFO's were also generated by CoreGEN. It was nice to be able to specify what kind of hardware resources each would consume, whether Block RAM, Distributed RAM or built-in units. Note that the built-in units may not simulate as easily; simply generate Block RAM modules for simulation and regenerate the cores with the same project file when moving back to synthesis.

10 Individual Contributions

10.1 Andrew Lau

My primary responsibilities for this project were the implementation of the vertex transformation pipeline and the integration and testing of the various parts of the pipeline. The first few weeks of the project were spent working closely with Alan to design the front end of the pipeline, specifically how to use GLSim to talk to the pipeline. This idea was quickly squashed as we found that GLSim was deprecated and had not been updated since 2002.

10.2 Alan Zhu

10.3 Nathan Wan

One Man Tools Team Web/SDK work PLB and System Design

11 Status and Future Work

11.1 Tool Chain

11.2 CPU Integration

11.3 Shader

A very interesting project would be to work on a graphics shader. OpenGL ES 2.0 relies primary on the shader to replace many fixed function units in the pipeline.

11.4 MPMC

11.5 vsync Timing

12 Class Impression/Improvements

12.1 Tool Chain Frustration

12.2 Too much Independence / No Feedback