OpenGL Accelerator

Tom Cherry, Eric Rippey, and Alvin Li

Introduction

Graphics cards are a complex and essential part of modern computing. The task of creating a graphics accelerator is one which should not be undertaken without an understanding of the scope of the project, which, if done correctly will produce very interesting and tangible results.

Today, there are two dominant systems for creating three-dimensional graphics. These are Direct3D and OpenGL. Direct3D is the portion of Microsoft's DirectX API which is used for three-dimensional graphics. It is intended to be a thin layer between a user space application and the drivers of the underlying graphics hardware. Being a part of DirectX, it is tied to Microsoft's platforms, running on versions of Windows since Windows 95, and on Microsoft's Xbox and Xbox 360 systems. The only known indepent implementation is in Wine, whose implementation is incomplete.

OpenGL is an industry standard for two, and more prominently, three-dimensional graphics which was originally created by Silicon Graphics in 1992. Since that time, it has been widely used in diverse areas of computer graphics, from its original application high-end graphics workstations to recent work on a subset designed for smartphones.

The widespread adoption of OpenGL, diversity of its implementations, and the existence of usable subsets meant that OpenGL was a better match for our purposes than Direct3d. We decided to implement a subset of OpenGL.

Early Work

Having decided on the basic graphics API, began on In creating a graphics The creation of a graphics accelerator Our project was to implement a subset of OpenGL. Classically, graphics cards have been implemented by having a bunch of different pieces of hardware, each of which does a specific thing in a pipeline. The previous group that had attempted to build a graphics card in this class had created a fixed-function pipeline, and from their report appeared to have made everything work together except for some bug that they would have been able to stomp had they been able to have another few days at the end of their project. Also, seeing material such as [Knutsson05], in which a masters student designed two different pipelines suggested the viability of this approach.

Knowing this history, it appeared that using a fixed-function pipeline would be a good way to go. In fact, we started to create one in Verilog. Going in to this though, we knew that there wouldn't be enough room on the FPGA to do a fixed-function pipeline with a bunch of floating point units. We decided to take the advice of the previous group to do this, and use fixed-point, as this is much smaller than floating point.

A decision to use fixed point does not fully describe the number format. There are two major things to decide about fixed point numbers. These are the overall size of the number, and the number of bits to the right of the decimal point. At first, we thought we were going to just use the same format that the old group used. However, we decided that this was an issue worth exploring. For this reason, we decided to run some tests to determine workable numbers. The way that we did this was to create a software version of OpenGL and then started changing the number format that was used in the different parts of the software. The order in which pieces were implemented was to start at the last part, with this lowest level piece being the rasterizer, and moving up in the stack towards the user. The results

were encouraging until got to the depth buffer. At that point, the fixed point numbers failed to operate correctly, and could not be prevented from producing a colorful cascade of miniature reflections scattered about the screen.

Fixed-point Numerical Operations

While we were working on the software implementation to determine the best format of the fixed point numbers, we really thought that there were going to be used, so started building parts that would be used in it. The first part that was built was a unit to do matrix multiplication. This was a single part that was very flexible, having parameters to adjust for the number of rows, the number of columns, and for the two types of variation of fixed point numbers. It was made to try to take up less area of the board over being fast, specifically to use on the the eight built-in multipliers of the Virtex II. Because used the straightforward calculation by the definition, took cubic time to produce results.

Additionally, there was a sqrt unit created which iteratively would refine the result through doing multiplies. Took O(n) time where n is the number of bits of the result. Also division was started, which followed a similar algorithm to the square roots, in which each bit would be determined iteratively.

There was also did some initial research on implementing trigonometric functions. Result seemed to be that the most straightforward way to implement them would be to start with sine, which would be a lookup table for a portion of its range, and then having flipping and shifting of the range for numbers outside of it. To implement this, one basically needed to know the format for the fixed-point numbers, so this was postponed until it turned out to be irrelevant.

Exploration with glxgears

One other interesting question to ask during the design phase was "exactly what subset of OpenGL are we going to use?" One way that we had to answer that we "just enough to make some solids move around on screen." It was also an interesting question to ask "so what is it that we're going to render?" On systems running the X windows system, there is often a 3d demo program called glxgears. The glxgears demo consists of a scene rendered onscreen with three gears that mesh as they spin, and can be rotated by the user by using the arrow keys.

The first step was to intercept all the calls to OpenGL and the glut library. Once the calls were intercepted, a system to record and play back the calls to the API was created. This allowed one to characterize the usage of the API and also to examine what happens when certain calls are ignored. For example, we knew that we didn't want to bother implementing lighting, and we found the if the calls to the lighting functions were simply ignored, everything else would continue to operate as normal.

Another application for the intercepted calls to the OpenGL library is to run a second display based at the same time that the primary is running. A program was written such that a seperate window could be created which would do the same thing as the original program's window.

This work would also allow one to do something remotely, which would mean that would be able to have a general purpose program on a computer, and then send the information to the FPGA for rendering. The obvious application of this would be to allow any program using the X11 OpenGL bindings to send drawing information to the FPGA as it ran. The way that this would have worked is that the program would just be recompiled to include an alternate version of the OpenGL library. This wasn't done as it was not a priority, nor was the implementation on the FPGA sufficiently fast to make

running standard applications on it interesting.

A static count of the number of uses of different library calls from glxgears was also created, along with the types of arguments each used. This was used to create an estimation of the bandwidth required for transferring information to the FPGA.

Return type	Frequency	name	Argument 1	Argument 2	Argument 3	Argument 4	Argument 5	Argument 6
	6	glBegin	int					
	3	glCallList	int					
	1	glClear	int					
	3	glDeleteLists	int	int				
	2	glDrawBuffer	int					
	5	glEnable	int					
	6	glEnd	n/a					
	3	glEndList	n/a					
	3	glFrustum	double	double	double	double	double	double
int	3	glGenLists	int					
	4	glGetString	int					
	1	glLightfv	int	int	const float*			
	4	glLoadIdentity	n/a					
	3	glMaterialfv	int	int	const float*			
	6	glMatrixMode	int					
	3	glNewList	int	int				
	7	glNormal3f	float	float	float			
	6	glPopMatrix	n/a					
	6	glPushMatrix	n/a					
	6	glRotatef	float	float	float	float		
	2	glShadeModel	int					
	2	glTranslated	double	double	double			
	4	glTranslatef	float	float	float			
	28	glVertex3f	float	float	float			
	1	glViewport	int	int	int	int		
XvisualInfo*	2	glXChooseVisual	Display*	int	int*			
GLXContext	1	glXCreateContext	Display*	XvisualInfo*	GLXContext	Bool		
void	1	glXDestroyContext	Display*	GLXContext				
bool	1	glXMakeCurrent	Display*	GLXDrawable	GLXContext			
void	1	glXSwapBuffers	Display*	GLXDrawable				

Note: Enums, size_blah, etc. have been called int.

Implementation

OpenGL

We chose to implement enough features from the OpenGL library to support drawing of triangles, which are the very basic and most essential OpenGL primitive and all of the essential transformations. It was of course impossible to implement a full set of OpenGL features due to both time and the constraints of the FPGA. In the table below, we detail which OpenGL commands we implemented and what they are used for.

glBegin(GLenum mode) and glEnd()

- These functions transition OpenGL into a state where drawing commands can be issued.
- Specifying mode will tell OpenGL how to arrange the following calls to glVertex.
- Our implementation only supports the mode GL_TRIANGLES.
- Within the drawing state, matrix transformation functions will throw exceptions.
- Outside of the drawing state, primitive drawing functions will throw exceptions.

glVertex3f(GLfloat x, GLfloat y, GLfloat z)

- This function is one of many that specifies a vertex coordinate when drawing a primitive.
- It tells OpenGL that the next vertex to be draw is at coordinate (x, y, z).

glColor3f(GLfloat red, GLfloat green, GLfloat blue)

- This function sets the current color of the renderer to the rgb color described in the input.

- All vertices drawn will be drawn with the current color of the renderer.
- It accepts floating point values between 0.0f and 1.0f for each red, green, and blue.

glClear(GLbitfield mask)

- This function accepts 3 arguments for the corresponding buffers it can clear.
- It can clear the depth, color, and stencil buffers.
- In our implementation we only have depth and color buffers, so the stencil bit is ignored.

glMatrixMode(GLenum mode)

- This function instructs OpenGL which matrix is the current matrix.
- Subsequent calls to transformation functions will modify only the current matrix.
- The 4 possible matrices are the modelview, projection, color, and texture matrices.
- In our implementation we only have the modelview and projection matrices.

glLoadIdentity()

- This function loads the identity matrix into the current matrix.

glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
GLdouble top, GLdouble nearVal, GLdouble farVal)

- This function specifies the frustum of the 3D world.
- A frustum is the 3D region that is visible on screen.
- In OpenGL, it is a clipped pyramid.

gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)

- This is a function from the OpenGL utility library, glu.
- It is a wrapper around glFrustum and converts the more meaningful values specified as a field of view and aspect ratio to the values required for the appropriate frustum call.

glViewport(GLint x, GLint y, GLint width, GLint height)

- This function specifies the location and size of the device on which OpenGL renders.

glTranslatef(GLfloat x, GLfloat y, GLfloat z)

- This function modifies the current matrix such that it is translated to the coordinate (x, y, z)

glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)

- This function rotates the current matrix angle degrees around the vector (x, y, z)

glScalef(GLfloat x, GLfloat y, GLfloat z)

- This function scales the x, y, and z values of the current matrix by x, y, z.

Other functionality

The above OpenGL commands simply specify what primitives should be drawn and the world in which they are to be drawn, and there are other functions essential to OpenGL that need to be described. The two important ones, without which it would be impossible to draw a primitive, are the function that projects a vertex from object coordinates to window coordinates and the function that rasters a primitive given its window coordinates. We implement both of these functions in our project.

Coordinate projection happens in 4 basic steps. First object coordinates are multiplied by the model-view matrix which produces eye coordinates. This transformation translates coordinates to their position, size, and rotation in the world. In the second step, eye coordinates are multiplied by the projection matrix which produces clip coordinates. This transformation puts the eye coordinates into the current perspective, and are called clip coordinates because at this point, vertices outside of the frustum can be clipped. Next, the coordinates are normalized into the range 0.0f..1.0f in preparation for the final stage, where they are multiplied by the viewport to produce window coordinates, which is the final location on the screen for a vertex.

Rasterization is the process of filling in the pixels of a primitive. Our implementation draws triangles using barycentric coordinates. The idea behind barycentric coordinates is that there exists the the following equation for a given v_0 v_1 and v_2 that correspond to the 2D screen coordinates which describe the triangle to be drawn.

$$x = \alpha v_0 + \beta v_1 + \gamma v_2 \text{ with } \alpha + \beta + \gamma = 1$$

This equation guarantees if each of α, β, and γ are in the range 0.0f..1.0f then the pixel x will be within the triangle and if α, β, and γ are not in the range 0.0f..1.0f then the pixel x is not within the triangle. The values α, β, and γ are computed as follows.

$$\alpha = f_{12}(x, y) / f_{1,2}(x_0, y_0)$$

$$\beta = f_{20}(x, y) / f_{20}(x_1, y_1)$$

$$\gamma = f_{01}(x, y) / f_{01}(x_2, y_2)$$

where in all cases

$$f_{ab}(x, y) = (y_a - y_b)*x + (x_b - x_a)*y + x_a*y_b - x_b*y_a$$

To use these equations in rasterization, we compute a bounding box around the triangle by determining the minimum and maximum x and y coordinate. We then iterate over each of these coordinates and compute the α, β, and γ values at that coordinate. If all of the values are within the range 0.0f..1.0f then we know the pixel lies within the triangle. We then take advantage of the fact that barycentric coordinates are able to be used for interpolation to interpolate the depth of the 3 z values at that pixel using the following formula.

$$depth = \alpha z_0 + \beta z_1 + \gamma z_2$$

Once we have the depth value of a pixel within the triangle, we check our depth buffer and see if we are closer to the screen than the last pixel drawn. If we are closer, then we finally know that we should draw. To determine the color, we once again interpolate with barycentric coordinates using the following forumla.

$$c = \alpha c_0 + \beta c_1 + \gamma c_2$$

We compute this for each color, red, green, and blue, and write the color to the framebuffer and the depth to the depth buffer. Psuedocode for the entire algorithm we use follows below.

xmin = min(x_0, x_1, x_2)

xmax = max(x_0, x_1, x_2)

ymin = min(y_0, y_1, y_2)

ymax = max(y_0, y_1, y_2)

for i in (xmin..xmax)

    for j in (ymin..ymax)

        if (i, j) are within viewport

            $\alpha = f_{12}(i, j) / f_{1,2}(x_0, y_0)$

            $\beta = f_{20}(i, j) / f_{20}(x_1, y_1)$

            $\gamma = f_{01}(i, j) / f_{01}(x_2, y_2)$

            if (α, β, γ) are within (0.0..1.0)

                depth = $\alpha z_0 + \beta z_1 + \gamma z_2$

                if (depth $<$ depthbuffer[i][j])

                    red = $\alpha r_0 + \beta r_1 + \gamma r_2$

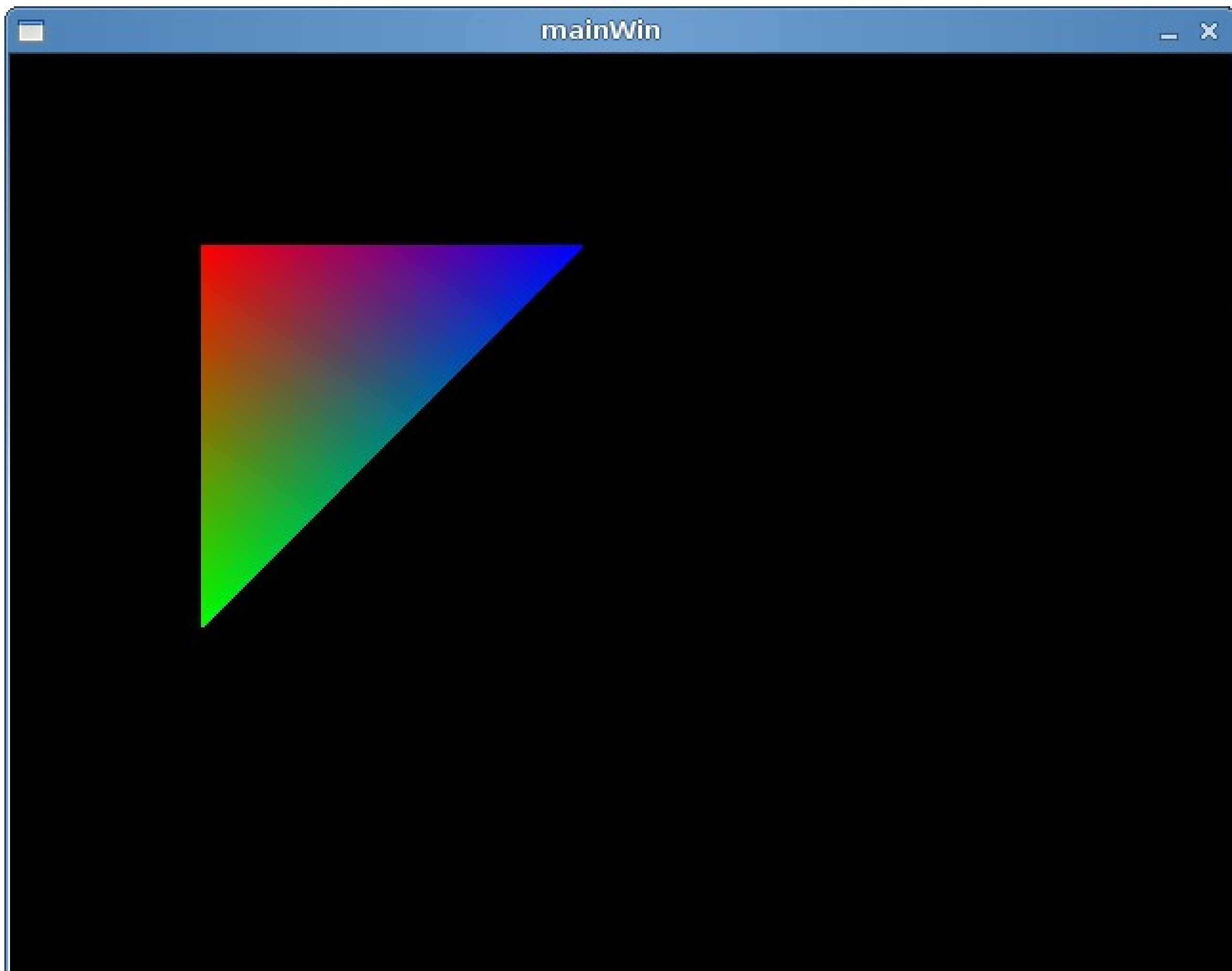
                    green = $\alpha g_0 + \beta g_1 + \gamma g_2$

                    blue = $\alpha b_0 + \beta b_1 + \gamma b_2$

                    framebuffer[i][j] = {red, green, blue}

                    depthbuffer[i][j] = depth

An example output of this algorithm is below.



Design methodology

Since we knew we were going to write every algorithm from scratch and we knew testing algorithms in verilog is harder than modeling them in a programming language and testing them there, our approach was heavy into modeling code in C to start and then testing it in verilog. Our testing platform was a C program that implements the entirety of what we planned on implementing. We wrote this from back to front. We started with the rasterization algorithm and tested it with by using screen coordinates and colors and viewed that it successfully drew a triangle in the correct position with the correct colors on

the screen. We then progressed closer to the OpenGL interface by writing the algorithms that transformed matrices and the coordinate transform algorithm. We tested this by writing a sample program that rendered two color pyramids and rotated them in place on the screen. It was a basic example that utilized a majority of the functions we implemented.

The next phase after we had a working model and decided that our approach with the project design was to be a processor running microcode was to both create this microcode and test it. To be able to write this microcode and evolve it parallel to designing the processor, we wrote a simple simulator in C that was capable of processing the microcode. We also created a small script in ruby which would take a microcode program and output arrays of microcode in C, raw text, and vhdl for use of making brams. The microcode was then written as plaintext. This phase of the design was particularly interesting, because if the output was incorrect after testing, the problem could be either with the microcode, assembler, or processor simulator. In time when our tests worked correctly, we were able to solidify our microcode and then move onto the verilog processor design. At this point we were able to dump registers, the depth buffer, and the framebuffer from our simulator and compare them to the output from modelsim. We wrote small test programs and verified our processors functinoality.

At this point, we had intended to then run large test programs through, for example any of the full microcode programs, however we were running out of time in the class. We also had difficulties wiring up the processor design to the memory, since the framebuffer and depth buffer are both too big to fit into bram and accessing SDRAM from verilog was not trivial. We had planned on using the processor as a powerPC peripheral and having the powerPC push down commands to the processor in terms of calling different microcode programs, and having the processor ask the powerPC whenever it needed to write to or read from SDRAM. We made effort down this path, however we were running out of time with the class and still did not have a solid demo program for the final week.

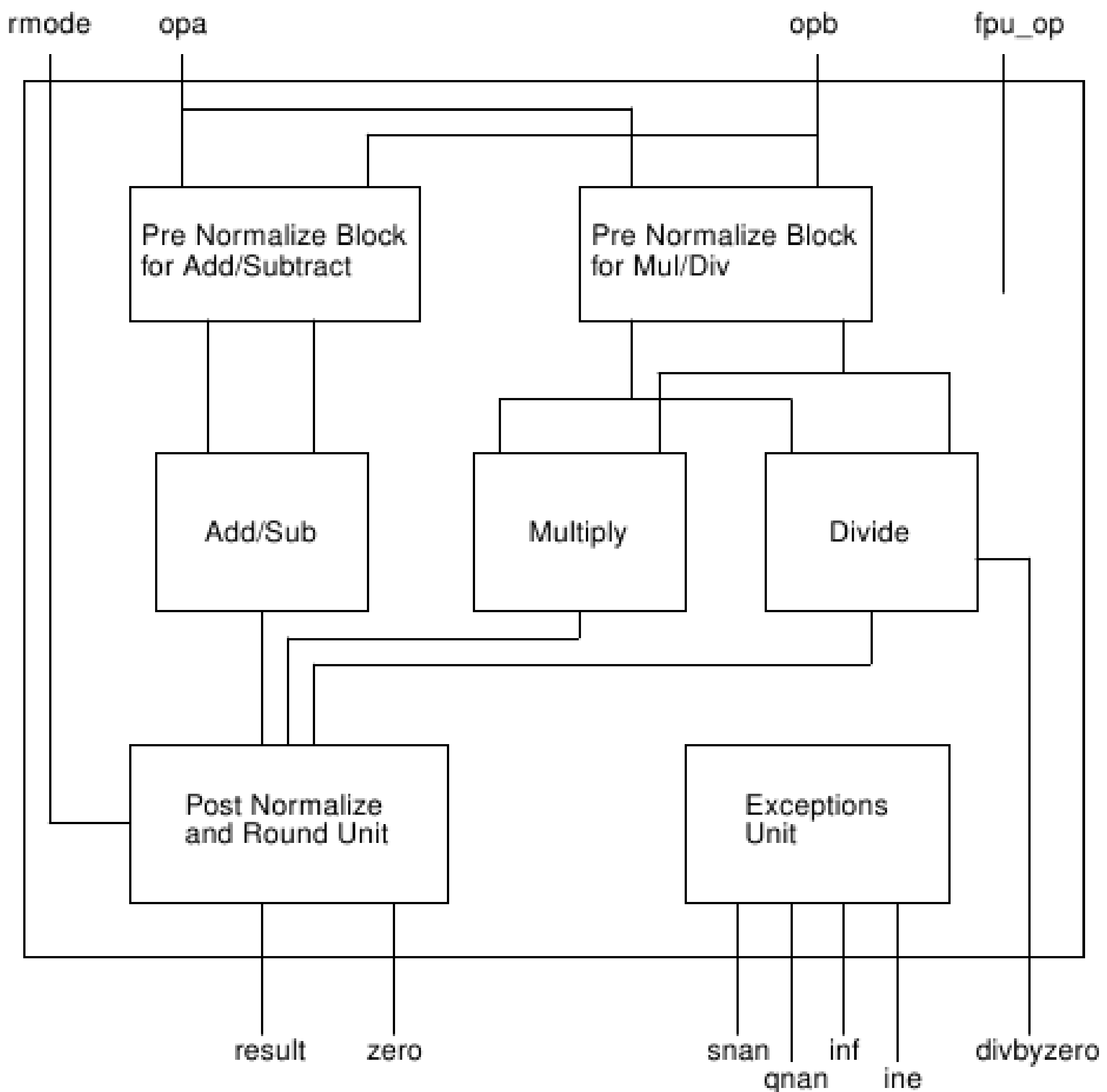
To be able to have a final demo, we resorted to using the same simulator that we used to verify the microcode on the powerPC. Our final project was almost exactly as we had first specified with the difference that the simulator running on the powerPC was the aforementioned “processor” instead of a verilog peripheral. The only other change is that the performance benefits from a VLIW design were unachievable, since we were not able to control 4 FPUs in parallel from the simulator and instead each floating point operation was handled separately.

The Floating Point Unit

As stated in the previous section of the report, we found that fixed point caused a lot of problems with our design. We decided to find an open source floating point unit written in Verilog instead and use that to do the necessary floating point arithmetic.

We found a 100% IEE754 compliant Verilog FPU on opencores.org written by Rudolf Usselmann. You can find the source at this url: http://www.opencores.org/project,fpu

Below is a block diagram of the FPU provided in the documentation written by Rudolf Usselmann.



This FPU takes 4 cycles perform an operation, it supports several different rounding modes and has an IEEE754 compliant exception unit. We decided there wasn't a need to change the rounding mode and that the rounding mode will not significantly affect the operation of our system. To simplify operation of the FPU, the rounding mode is permanently set to “round to nearest even.”

The FPU was initially unsynthesizable. This was partly due to the implementation of the Divide submodule shown in the diagram above, which just used an assign statement with the '/' operator. After we changed the divider into a shift and subtract divider, synthesis was still unsuccessful. For some unknown reason, the original Verilog code had some unsynthesizable “#1” delay statements in the code. After we removed those, the FPU is synthesizable. The synthesized FPU took up approximately 20% of the available LUT's on the Virtex 2 FPGA, and we clocked it at 50 MHz.

To test the FPU, our first step was to use the test unit that came with the FPU. The test unit is a basic comparison with known value tester that used a large number of test vectors generated by a script (approximately 14 million). We had to make some changes to the tester before it could run as well. There were some extraneous references to non-existent test files within the test unit. We used ModelSim to run the test unit on the generated test vectors. After all of those test vectors passed, we were pretty confident that the functionality of the FPU on its own was correct.

The next step was to make the FPU interface with the PowerPC core on the FPGA board. We wrote the standard interface code that allows the PowerPC to access the inputs and outputs of the FPU as memory-mapped registers through the PLB. We then wrote some C code that runs on the PowerPC to test the operation of the FPU on the board and confirmed that it works on the board.

VLIW Processor

When the decision was made to abandon the fixed-function pipeline, the alternative was to build a microprocessor with firmware that would run OpenGL. We decided to build a VLIW processor because it could be done in a relatively simple way and our application called for heavy floating point and lots of registers. We also wanted a fairly orthogonal instruction set for easier coding and debugging.

The decision to keep things simple meant that the software meant that we decided not to have a stack, but to have a very large register file to compensate, which would be fine since it would be for a dedicated purpose. Also, many things that we found inessential were left out, such as memory protection, shifts, rolls, and, or instructions, and interrupts. There is no have pipelining or branch prediction. It is also a Harvard architecture machine; all the instructions are in a ROM which is put on the FPGA with the hardware.

The design included four floating point operations per cycle, which, it turned out while writing the microcode was not the full extent of what could be parallelized.

Instruction Format

There are 128 bits per instruction, which are allocated as described below.

Bits	Meaning
0-4	Logic unit 0, operation
5-11	Logic unit 0, destination register
12-18	Logic unit 0, source register 0
19-25	Logic unit 0, source register 1
32-36	Logic unit 1, operation
37-43	Logic unit 1, destination register
44-50	Logic unit 1, source register 0
51-57	Logic unit 1, source register 1
64-68	Logic unit 2, operation
69-75	Logic unit 2, destination register
76-82	Logic unit 2, source register 0
83-89	Logic unit 2, source register 1
96-100	Logic unit 3, operation
101-107	Logic unit 3, destination register
108-114	Logic unit 3, source register 0
115-121	Logic unit 3, source register 1

opcode	Meaning	opcode encoding	Destination register use	Value source	Allowed only in logical unit 0
fadd	Floating point addition	1	1	logical unit	0
fsub	Floating point subtraction	3	1	logical unit	0
fmul	Floating point multiplication	5	1	logical unit	0
fdiv	Floating point division	7	1	logical unit	0
fcmp	Floating point comparison	9	1	logical unit	0
f2i	Convert floating point to integer	11	1	logical unit	0
i2f	Convert integer to floating point	13	1	logical unit	0
imax	Calculate maximum of the input operands as integers	15	1	logical unit	0
imin	Calculate minimum of the input operands as integers	17	1	logical unit	0
iadd	Integer addition	19	1	logical unit	0
isub	Integer subtraction	21	1	logical unit	0
imul	Integer multiplication	23	1	logical unit	0
icmp	Integer comparison	25	1	logical unit	0
bg	Branch on greater than 0	27	0	n/a	1
bl	Branch on less than 0	29	0	n/a	1
be	Branch on zero	31	0	n/a	1
nop	No operation	0	0	n/a	0
yield	Halt	2	0	n/a	1
store32	Write word to memory	4	0	n/a	1
store8	Write byte to memory	6	0	n/a	1
load32	Read byte from memory	8	1	memory	1
lli	Load lower immediate	10	1	inst	0
lui	Load upper immediate	12	1	inst	0

Assembly Code

The code for the VLIW processor is written in a custom assembly language. This language bears some resemblance to Intel format x86 assembly. Each line is either blank, a comment, or a single instruction. Blank lines must contain only whitespace characters. Comment lines begin must with a semicolon and end with a newline character. Each instruction line consists of four operations, one for each logical unit.

There are some restrictions on the opcodes:

1. The same register should not be read and written by different operations during the same instruction.
2. More than one operation may not write to the same register during the same instruction
3. Certain opcodes are restricted to appearing in the zeroth position. This is mainly for instructions where it having more than one run at once could create logical contradictions not ameliorated by the restriction on register destinations, and for load32.
4. During the store8 operation, instruction number three will have one of its input values overridden by a value which is used to calculate the address of the store. For this reason, though it is possible, it would be unwise to use logical unit 3 while running store8.

For an assembly language, one must have an assembler. Ours is written in ruby and supports a variety of output formats. The simplest of these is output with the ".raw" extension, and consists of each instruction on its own line with each of the operations as a decimal number, separated by spaces. This exists for easy machine reading by scripts. The assembler also produces its output as C arrays, which makes for easy inclusion in the simulator. Yet another output type the assembler is capable of

outputting is Verilog modules. These are used to instantiate ROMs on the FPGA. The final output format supported is VHDL. The reason for a VHDL output mode in addition to the Verilog one is that the Xilinx tools do not support initializing BRAMs with Verilog. Therefore, when we attempted to move the instructions into a BRAM, it this became necessary.

Testing

The testing of the processor was mainly by writing a set of short test programs, one to test each of the instructions. All of the instructions besides loads and stores were shown to work.

Size-reduced version

When synthesized, the original processor took 159% of the space available on the FPGA. This was not entirely unforeseen For this reason,we had already started work on a version of the processor that would require less space on the FPGA.

The cut down version of the processor is very similar, though not identical from a programmer's perspective. All instructions which use units 1-3 become NOP, and jumps now target only every four instructions. The reason that the targets for branches become only every four instructions is so that jump targets do not have to be recalculated when porting code meant for the other architecture.

The similarity of the machine code, along with the restriction on reading and writing to the same register with different operations, meant that a binary to binary translation of the microcode was not only possible, but feasible and generally painless. The algorithm to translate to code for the reduced-size version is as follows: for each instruction in the original program, take the portion for logical unit three and output an instruction which consists of that portion moved to logical unit 0, and other logical units being send NOP. Next, treat the second operation in the same way, and the first, until the zeroth

operation is output. The reason that the operation for logical unit 0 is executed last is that otherwise, in the case of a jump, not all of the original operations would be executed.

Testing

The testing for the size-reduced version of the processor was very straightforward. Once the script to convert all the instructions to the new format was run, the same tools and method could be used to do the testing.

Final demonstration

Since our project was not implementing a game, we did not have a clear idea at the beginning of what our final would be. There are of course infinite demos possible with an OpenGL accelerator. We had written a small demo program for the in class demo on Wednesday which intended to show off our implementation by rotating a pyramid such as the one we used for testing on the screen. After the demo, we realized that only having a pyramid rotating does not fully show off the capabilities of our implementation, and doesn't give guests for the Friday demo anything interactive. To remedy this, we added keyboard input that would allow users to move, scale, and rotate two triangles on the screen. We accomplished this by issuing the following OpenGL commands.

For moving the pyramids right, left, up, down, closer, and further:

glTranslatef(1.0, 0.0, 0.0)

glTranslatef(-1.0, 0.0, 0.0)

glTrsnalatef(0.0, 1.0, 0.0)

glTranslatef(0.0, -1.0, 0.0)

glTranslatef(0.0, 0.0, 1.0)

glTranslatef(0.0, 0.0, -1.0)

For doubling and halving the size of the pyramids.

glScalef(2.0, 2.0, 2.0)

glScalef(.5, .5, .5)

For rotating the triangle at different speeds

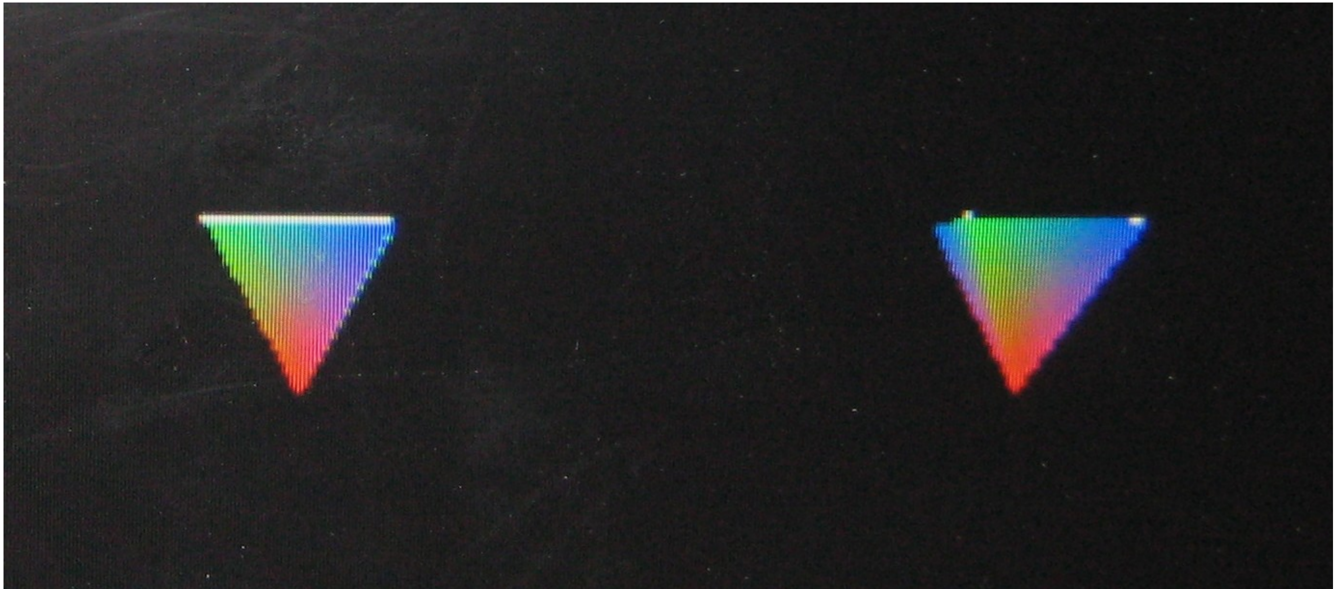
glRotatef(15.0, .5, .5, .5)

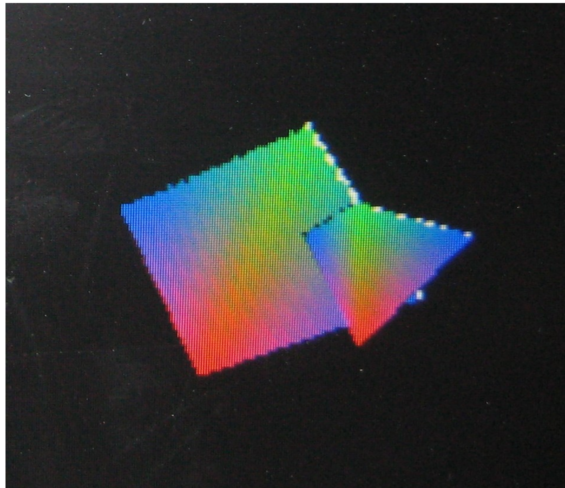
glRotatef(30.0, .5, .5, .5)

The demonstration program begins with two pyramids located left and right of the center of the screen. They can be instructed to rotate, and be moved in any direction independently. This allows for users to see the full capabilities of our implementation. For example, one can scale a pyramid to be large while moving it in back of the other pyramid to show two pyramids rotating on top of each other. One can also move the triangles close enough to each other to show off the depth buffer as they rotate without artifacts. We also included a feature to bring the pyramids back to the starting point in the case that the user would want to start over. This helped alleviate problems such as when a user scaled a pyramid too large and the frame rate dropped sufficiently low. One other problem that arose was that movements on rotating pyramids were relative to the pyramid and not relative to the screen. This was actually a problem in our demo program, where we would continually apply a new rotation instead of starting over at each step, and thus subsequent translations took were not relative to the screen.

Below are three images taken with a camera of our demo program. The first is a picture of the demo program after being rest, the second is a picture of our demo program with rotated pyramid on top of

each other. The third is a picture of our demo program with two pyramids, however with the smaller pyramid in the background.





Microcode

All of the algorithms we used became microcode programs. We ended up having around 700 lines of microcode, which is 2800 separate instructions after being packed into 4wide VLIW format. Below is an example of calculating the divisors for α, β, and γ during rasterization.

```
;calculate bottoms
isub vertex11 vertex21 temp0; isub vertex20 vertex10 temp1; imul vertex10 vertex21 temp2; imul vertex20 vertex11 temp3
;alpha a - b; alpha b - 1; alpha ab; alpha ba
imul temp0 vertex00 temp4; imul temp1 vertex01 temp5; isub temp2 temp3 temp6; isub vertex21 vertex01 temp7
;alpha (a-b)x; alpha (b-a)y; alpha ab-ba; beta a-b
iadd temp4 temp5 temp0; isub vertex00 vertex20 temp1; imul vertex20 vertex01 temp2; imul vertex00 vertex21 temp3
;alpha (a-b)x + (b-a)y; beta b-a; beta ab; beta ba
iadd temp0 temp6 atop; imul temp7 vertex10 temp4; imul temp1 vertex11 temp5; isub temp2 temp3 temp8
;alpha done; beta (a-b)x; beta (b-a)y; beta ab-ba
iadd temp4 temp5 temp2; isub vertex01 vertex11 temp3; isub vertex10 vertex00 temp6; imul vertex00 vertex11 temp7
;beta (a-b)x + (b-a)y; gamma a - b; gamma b - a; gamma ab
iadd temp2 temp8 btop; imul temp3 vertex20 temp0; imul temp6 vertex21 temp1; imul vertex10 vertex01 temp2
;beta done; gamma (a-b)x; gamma (b-a)y; gamma ba
iadd temp0 temp1 temp3; isub temp7 temp2 temp4; nop; nop
;gamma (a-b)x + (b-a)y; gamma ab-ba ; nop nop
iadd temp3 temp4 gtop; nop; nop; nop
i2f atop temp0; i2f btop temp1; i2f gtop temp2; nop
fdiv rf1 temp0 abottom; fdiv rf1 temp1 bbottom; fdiv rf1 temp2 gbottom; nop
```

This example contains comments which help us see what operations are done with each instruction. It also contains aliased registers, so that we can use meaningful names such as vertex00 to describe the x

component of the 1st vertex as opposed to everything being refered to as r#. This example code corresponds to the following code from our C model.

```
#define f(a, b, x, y) \
  ((GLfloat)((vertices[a][1] - vertices[b][1]) * x + \
(vertices[b][0] - vertices[a][0]) * y + \
vertices[a][0] * vertices[b][1] - \
vertices[b][0] * vertices[a][1]))

alphaBottom = 1.0f/f(1, 2, vertices[0][0], vertices[0][1]);
betaBottom = 1.0f/f(2, 0, vertices[1][0], vertices[1][1]);
gammaBottom = 1.0f/f(0, 1, vertices[2][0], vertices[2][1]);
```

Future Work

Our goal in this project was to implement a subset of the OpenGL graphics API, a large enough subset that would perform basic 3D graphics operations. As such, there is a lot of room for extension of the project. One of the main challenges would be to increase the frame rate of the processor. This would require either a move to a faster platform with more resources and/or implementing caching of accesses to the frame buffer and depth buffer, since the large number of memory accesses is one of the bottlenecks of our current implementation.

Other than pure performance improvements, there are several ways to expand the project. A large number of features of a typical commercial OpenGL processor are not supported by our current design, such as texture, shading, lighting, just to name a few. Implementing these features would add significantly to the complexity of the processor. Another idea that would work well is using multiple cores, since many graphics operations can and should be done in parallel. Dealing with the distribution of work and resources among several processors would be another interesting challenge to tackle. The final improvement would be to pipeline many of the operations in the processor. This would help increase performance and narrow the gap between a commercial GPU and our implementation.

Due to our constraints on time this semester, we were not able to implement these features. Future work on implementing an OpenGL processor on an FPGA could consider all of these potential design decisions.

Lessons Learned

In a semester long project like this, there can be a lot of good and bad decisions along the way. Our group was no exception, we made some good decisions and some bad ones that resulted in a lot of wasted time and effort. There are several things we wish we knew at the beginning of the project that could've helped us avoid those bad decisions. One of which is that switching your platform in the middle of the project is a bad idea. Although the Virtex 2 and Virtex 5 are similar, due to a lack of example code and example projects (since all the labs were done with the Virtex 2), figuring out how to do the same things on the Virtex 5 was difficult and time consuming. Ultimately, we decided to revert back to the Virtex 2 since we knew how to develop on it and it was familiar. If we had more had switched over earlier and were provided better documentation, we would have been successful with the Virtex 5. In future versions of this course, this shouldn't be a problem since the next class will probably

all be using Virtex 5's to start with.

And due to our switch back to the Virtex 2 late in the semester, we had to cut down on the scale of the hardware implementation. On the Virtex 5, we could easily fit 4 FPU's without going over 100% LUT utilization. The Virtex 2 has much fewer resources and this severely constrained our design space. So the lesson here is that you must decide on your platform early in the design process and stick with your decision. Changing your platform can have dire consequences on the success of your project. Sometimes, it's better to go with the platform you are more familiar with, rather than the one that has more resources or higher performance.

The next lesson we learned is that debugging hardware is very difficult, and can take an indefinite amount of time. If your group hopes to get a large and complex module in Verilog to function correctly, start debugging early. Also, don't debug by using the place and route in the Xilinx XPS, since this makes each fix/ test/ debug cycle last 40 minutes to an hour. Simulation is the way to go.

One of our best design decisions was to go with the floating point format instead of fixed point. This decision meant that we could just find an open source Verilog description of a FPU and use that, instead of writing our own custom arithmetic units in a non-standard fixed point format. Using FP also meant we didn't have to worry about loss of precision, and just made our lives easier in general.

Here are some more words of wisdom for future generations. Putting together a proof of concept for your project is a good idea. Although it seems like a waste of valuable time, half the time you'll realize that the way you were about to implement something doesn't actually work out and you'll save yourself a lot of grief. Next, If you haven't tested some part of your design, assume that it doesn't work. Finally, if you need to use something that is already extant, try to find source C code or Verilog for it. If you use need to integrate with something that is not entirely in your control, things get very difficult.

Lastly, we would like to complain about the fact that there doesn't exist a 64bit Linux version of the Xilinx tools, which prevented one of our group members from having the

Xilinx tools installed on his personal computer.

Tom Cherry's Personal Page

I did a large portion of the algorithms and their designs since I had the most experience with computer graphics. I developed most of the original C model from the mathematical operations from mesa 3D, developed code for rasterization from a paper describing the mathematics of barycentric coordinates and developed the code for coordinate projection by adapting the code from gluProject. I also then used this model and a fixed point library I downloaded from the Internet to explore the option of using fixed point arithmetic. This took in total around 20 hours at the beginning of the semester.

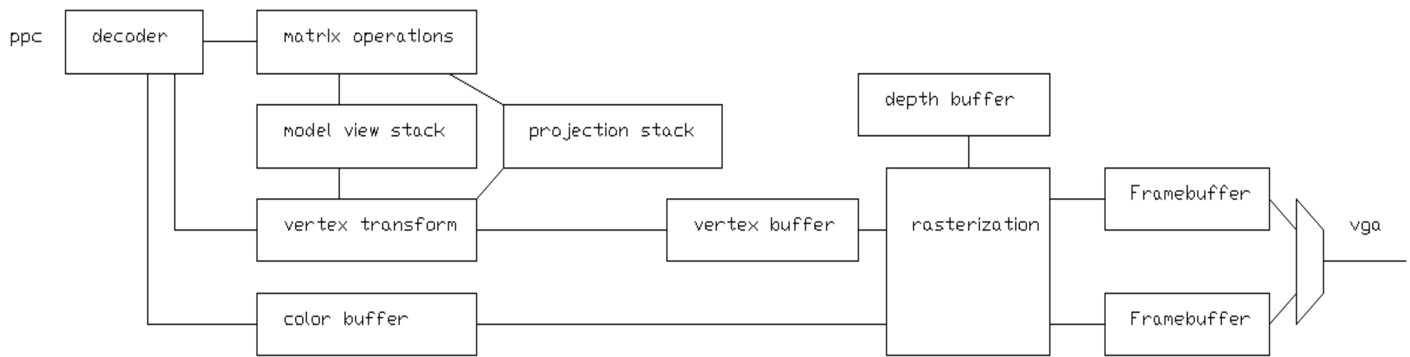
When the project took a turn towards microcode development, I created a majority of the microcode, the ruby parser script, and the C model processor simulator. I helped design the ISA, since it evolved as I wrote more microcode. The model processor came last, and was made to work with the already written microcode. Debugging the microcode took a substantial amount of time, and I had to often dump the registers at various stages in the execution and verify them by hand. It took roughly 20 hours for the first draft of the microcode. It took roughly 10 hours to originally create the simulator and assembler and do initial debugging. It took another 20 to 30 hours to fully debug the code before our

final demo.

I worked with my team on some of the labs and working with Xilinx to produce the final demo. I spent in excess of 20 hours working with just the FPGA and getting used to it's intricacies. The final demos on Wednesday and Friday took another 10 hours to create.

I thoroughly enjoyed getting to work with computer graphics on a low level. It was interesting to see how one goes about translating the coordinates inputted to OpenGL to the screen and rendering a triangle in the last step.

I do not have any complaints for the class other than I wish we had more time to work on the project. At the beginning of the semester, we spent a lot of time trying to figure out how we were going to implement the hardware, and had we of settled on a design earlier we would have likely made more progress. The switch to the Vertex 5 was also bad for us, since we had trouble getting it to work as we wanted and ended up switching back to the Vertex 2 with all of the time spent with the Vertex 5 lost. It would be excellent if labs for the Vertex 5 were written similar to the labs for the Vertex 2 in future semesters.



Individual Comments

Alvin Li

For the first week of the semester our group was deciding what kind of project we wanted to do. Originally we were thinking about making Duck Hunt with a light gun as input. But we couldn't find any good open source versions of the game code. We also explored other games such as Battle City and Wolfenstein 3D, but eventually decided on something that was more interesting technically, an OpenGL processor.

Unlike me, both Tom and Eric had at least a bit of exposure to computer graphics. I spent the first couple weeks familiarizing myself with the subset of OpenGL that we were planning on implementing. Next, I familiarized myself with the Virtex 2 FPGA's through reading and going through part of the

labs.

After the first design review, we decided to switch to the Virtex 5, which eventually turned out to be a bad decision. I spent several weeks trying to figure out how to figure out port our existing design over to the Virtex 5. However, because of the lack of good documentation or a tutorial, since the labs were all designed for the Virtex 2, I was unable to port most of the design. I did get the FPU to successfully synthesize on the Virtex 5 however.

Near the end of the semester we decided to give up on the Virtex 5 and switch back to the Virtex 2. At this point, the code Verilog code written for our processor was still not working. I helped Eric debug the Verilog code but we couldn't finish debugging in time for the final demo. Our group decided to go with a software version of the processor that used a hardware FPU for arithmetic operations.

Throughout the semester I also helped with the presentations, design reviews, writeups, etc.

I thought this class was very interesting since I got to learn interactively about FPGA's and computer graphics. Other than my limited experience with FPGA's in 240, I had no exposure to either of these topics. Although our final hardware design didn't work, I feel that I learned a lot about a how to design and implement a semester-long hardware project. If we had another week or two, we could've gotten the hardware processor debugged and working, although it's unclear whether there would be any significant performance boost by switching to the hardware version.

My main complaint about the class is that there wasn't enough documentation about dealing with the Virtex 5. I hope that in later iterations of the class the labs will be using the Virtex 5, and there will be substantial documentation on the Virtex 5, and the students get complete versions of the Xilinx ISE 11

tools and not have to worry about the trial version running out of time.