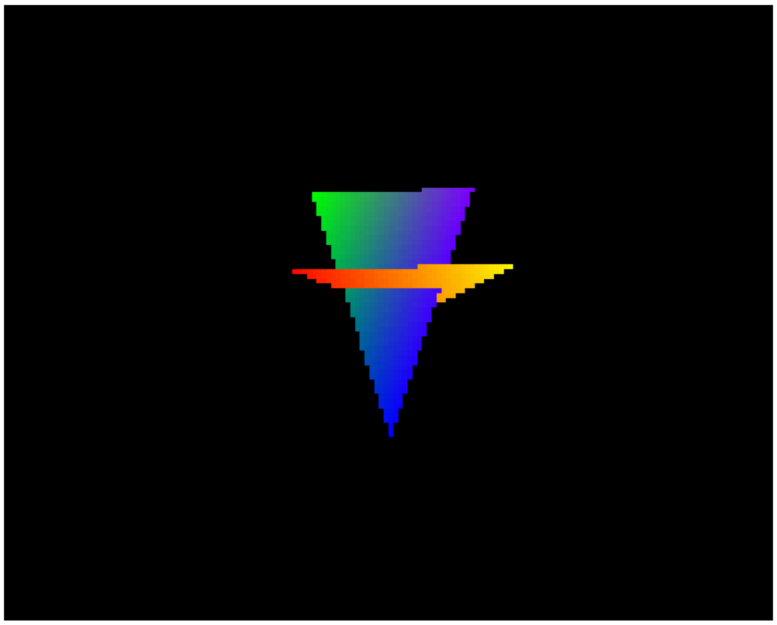

rayGPU User Manual

Ray Wendt

Revision 1.0

June 15, 2021

Revision History

Date	Version	Author	Comment
June 15, 2021	Rev. 1.0	Ray Wendt	Initial revision.

Contents

1 System Overview 2

2 Interface 3
2.1 Vertex Shader Constant Register 4
2.2 Fragment Shader Constant Register 4
2.3 Vertex Data Size Register 4
2.4 Vertex Data Base Address Register 5
2.5 Vertex Count Register 5
2.6 System Control Register 5

3 Floating-point Operation Details 5
3.1 Floating point format 5
3.2 Addition 6
3.3 Multiplication 6
3.4 Reciprocal 6

4 Shader Processor 6
4.1 Instruction format 6
4.2 Operand format 6
4.3 Destination format 7
4.4 Registers 7

5 Rasterize Algorithm 8

6 Display Output 8

7 Application Information 9

8 Timing 9

A Module Operation Details 10
A.1 Shader Processor 10
A.2 Vertex Shader Control 10
A.3 Rasterizer 10
A.4 Fragment FIFO 11
A.5 Fragment Shader Control 12
A.6 Framebuffer and Display Control 12
A.7 Top-level Control 13

B Testing and Supporting Code 13
B.1 Entire system testing 13
B.2 Floating point math testbenches 13
B.3 Display output test 14
B.4 Triangle demo MCU code 14

C Code Structure 15

1 System Overview

The system implements a 3D graphics pipeline with programmable vertex and fragment shaders and a fixed-function rasterizer. Vertex data representing triangles is loaded from RAM and processed by the vertex shader program. Shaded vertices are loaded in groups of 3 as triangles into the rasterizer, which converts the vertex coordinates to clip space, generates fragments inside the triangle bounds, interpolates depth and vertex attributes,

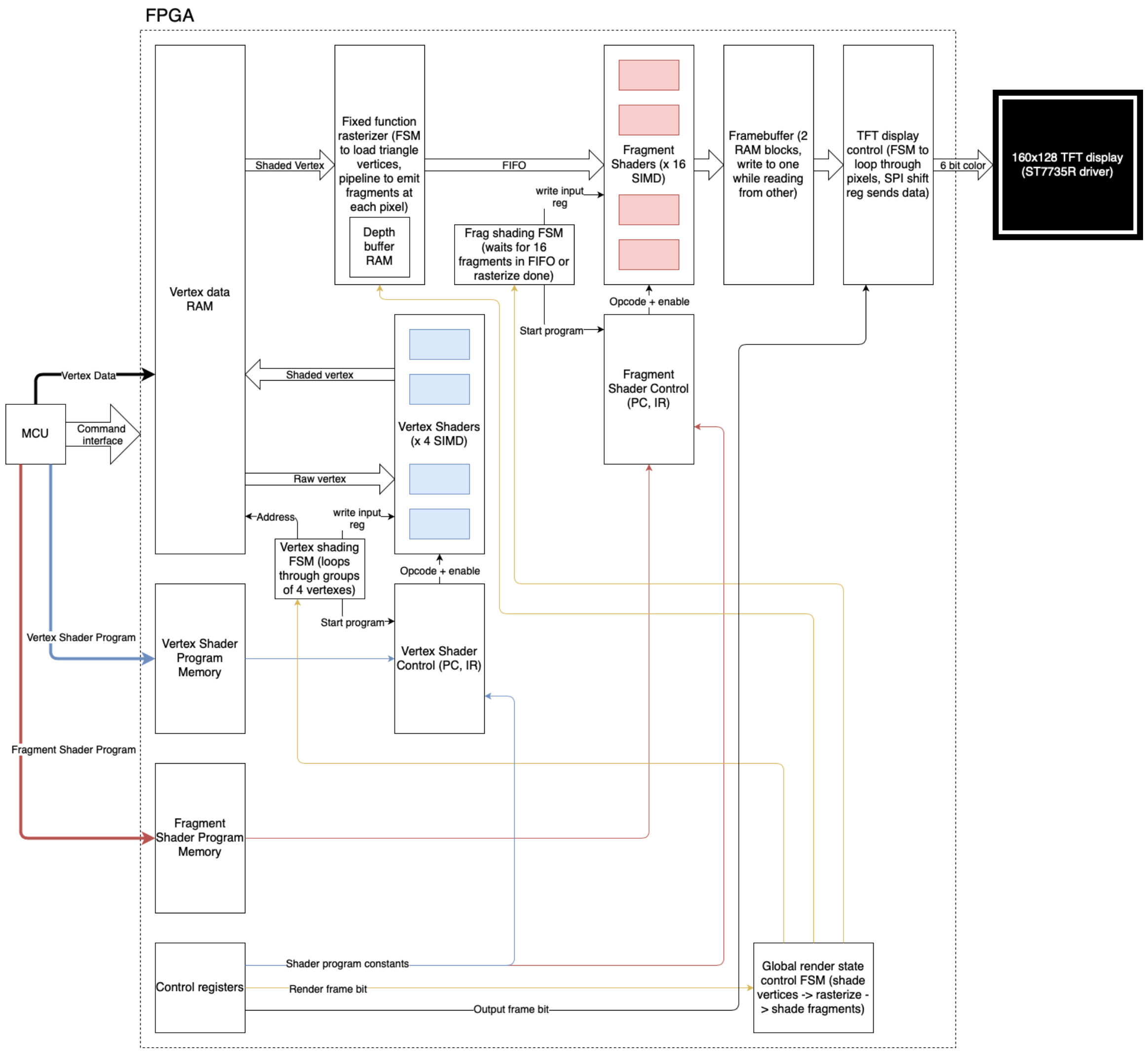
FPGA
Vertex data RAM
Shaded Vertex
Fixed function rasterizer (FSM to load triangle vertices, pipeline to emit fragments at each pixel)
Depth buffer RAM
FIFO
write input reg
Frag shading FSM (waits for 16 fragments in FIFO or rasterize done)
Fragment Shaders (x 16 SIMD)
Framebuffer (2 RAM blocks, write to one while reading from other)
TFT display control (FSM to loop through pixels, SPI shift reg sends data)
6 bit color
160x128 TFT display (ST7735R driver)
Opcode + enable
Start program
Fragment Shader Control (PC, IR)
Vertex Data
MCU
Command interface
Shaded vertex
Vertex Shaders (x 4 SIMD)
Raw vertex
Address
write input reg
Vertex shading FSM (loops through groups of 4 vertexes)
Opcode + enable
Start program
Vertex Shader Program
Vertex Shader Program Memory
Vertex Shader Control (PC, IR)
Fragment Shader Program
Fragment Shader Program Memory
Control registers
Shader program constants
Render frame bit
Output frame bit
Global render state control FSM (shade vertices -> rasterize -> shade fragments)

Figure 1: System block diagram

and performs a depth test against depth values stored in memory (the depth buffer) to determine whether to emit that fragment. Emitted fragments are processed by the fragment shader program which outputs the final pixel color, stored in the framebuffer. The framebuffer is double-buffered to allow writing to one buffer while the other is output to the display.

2 Interface

The GPU is controlled by a bus with 16-bit address and 16-bit data, which enables the user to write to system registers and memory. Data is written on rising edges of the write clock input. All registers are 16 bits wide, as well as vertex data memory. Shader program memory is 48 bits wide, so 3 writes are required per instruction, with the two high 16-bit words first and then the low 16-bit word, which triggers the write to memory. Frames are output to a 128x160 pixel display driven by an ST7735R chip, using its 4-wire serial interface, and targetting the Adafruit 1.8" breakout board but other displays with the same driver may also work.

Signal name	Bit width	I/O	Description
clk	1	Input	System clock. Assumed 100 MHz but can be reconfigured in Xilinx PLL IP core.
rst	1	Input	Active low system reset.
PLL_rst	1	Input	Active high PLL reset. Separate from `rst` because system reset is synchronous with PLL output.
addr	16	Input	System address bus, references register or memory to write to. Details in Table 2.
data_in	16	Input	Data to write to register referred to by system address bus.
wr_clk	1	Input	Clock for writes from system bus; write of `data_in` to `addr` occurs on rising edge.
test_out_1	1	Output	Test LED output for troubleshooting. Sticks high once a non-zero color is written to any pixel in the framebuffer.
test_out_2	1	Output	Test LED output for troubleshooting. Sticks high once the system reaches the rasterization and fragment shading state.
test_out_3	1	Output	Test LED output for troubleshooting. Set high (combinationally) when address input is 0xDEAD.
test_out_4	1	Output	Test LED output for troubleshooting. Set high (combinationally) when data input is 0xBEEF.
MOSI	1	Output	Serial SPI data output to display.
DCX	1	Output	Command/data select output to display. Low for commands and high for data.
SCK	1	Output	SPI clock output to display. Runs at 10 MHz. Idles low and writes on rising edges.
CSX	1	Output	Active low chip select output to display.

Table 1: System pin interface

2.1 Vertex Shader Constant Register

Vertex shader constant registers 0 to 3 provide constant values for the vertex shader program, where each register is a 4-element vector of 16-bit floats.

2.2 Fragment Shader Constant Register

Fragment shader constant registers 0 to 3 provide constant values for the vertex shader program, where each register is a 4-element vector of 16-bit floats.

2.3 Vertex Data Size Register

Indicates the number (minus one) of 4-element vectors stored sequentially in RAM for each vertex. The same number of outputs from the vertex shader is stored in RAM once vertex shading is complete. Inputs are loaded into input registers 0 to (vertex data size) and outputs are saved from output registers 0 to (vertex data size) for each processor. Note the minimum data size is 1 (register value 0), since vertices always require at least coordinate output from output 0.

Address	**Register or memory location**
00100aaaaaaaaaaa	Address a in vertex data memory.
00101xaaaaaaaaii	Address a in vertex program RAM, 16-bit word i (where 2 is most significant and 0 is least). Writes to words 1 and 2 write to a holding register and write to word 0 writes the holding register plus new data into RAM.
00110xxxxxxxxiijj	Element j in vertex shader constant i.
00111xxxxxxxxxx00	Vertex data size register.
00111xxxxxxxxxx01	Vertex data base address register.
00111xxxxxxxxxx10	Vertex count register.
01001xaaaaaaaaii	Address a in fragment program RAM, 16-bit word i (where 2 is most significant and 0 is least). Writes to words 1 and 2 write to a holding register and write to word 0 writes the holding register plus new data into RAM.
01010xxxxxxxxiijj	Element j in fragment shader constant i.
100xxxxxxxxxxxxx	System control register.

Table 2: System address list

2.4 Vertex Data Base Address Register

Indicates the address of the first vertex data in vertex RAM. Note that vertex shader outputs are written starting at this address with the high bit flipped, so not more than 1024 data entries should be written starting at the base address.

2.5 Vertex Count Register

Indicates the number (minus one) of vertices which are stored in RAM and should be processed. This value should be a multiple of 3 (minus one); otherwise when processing triangles, data will be loaded past the end of the data written by the user in RAM to complete the last triangle.

2.6 System Control Register

The system control register is used to trigger the beginning of processing events. A write to bit 0 clears the active depth and frame buffers, then begins rendering the current frame. A write to bit 1 swaps the framebuffers and begins output of the now inactive framebuffer to the display. These two processes can occur at the same time due to the double buffering of frames, but attempting to trigger each while it's already in progress has no effect.

3 Floating-point Operation Details

3.1 Floating point format

Bit 15	Bit 14 to 10	Bit 9 to 0
Sign (s)	Exponent (e)	Mantissa (m)

$$f = (-1)^s 2^{e-15}(1 + 2^{-10}m)$$

Figure 2: 16-bit float format

The system uses half-precision 16 bit floating point values globally, based on but not entirely implementing the IEEE-754 format. The high bit represents the sign, the next 5 bits represent the exponent in excess-15 format, and the low 10 bits represent the mantissa with an implied leading 1 in (unsigned) Q1.10 format. Positive and negative infinity are represented with exponent 16 (all exponent bits are 1) and positive and negative 0 are represented with exponent -15 (all exponent bits are 0). All mantissa bits are set to 0 for infinity and zero edge case values; values with exponent 16 or -15 and nonzero mantissa bits are meaningless and give unspecified outputs if input to the system. As a consequence, NaN and denormal values are not supported.

3.2 Addition

Addition is performed to the precision of the addend with larger exponent, with no rounding. That is, any bits in the mantissa of the addend with smaller exponent which are below the lowest mantissa bits of the addend with larger exponent are truncated and ignored. Sums with magnitude greater than or equal to 2^{16} overflow to positive or negative infinity as appropriate. Likewise, sums with magnitude less than 2^{-14} underflow to positive or negative zero as appropriate. Sums which are exactly 0 always result in positive 0 output. If the first addend is positive or negative infinity, the result is equal to the first addend no matter the value of the second. Otherwise, if the second addend is positive or negative infinity, the result is equal to the second addend.

3.3 Multiplication

Multiplication is performed to full precision allowed by the mantissa bits, also with no rounding. Overflow and underflow are handled the same as addition, except products which are exactly 0 still take the sign corresponding to the product of the signs of the multiplicands. A product with 0 is always magnitude 0 even if the other multiplicand is infinity. Otherwise, if one multiplicand is infinity, the product is magnitude infinity and also has the sign corresponding to the product of multiplicand signs.

3.4 Reciprocal

Reciprocals are calculated by negating the exponent and using a lookup table on the first 6 bits of the mantissa, so the operation does not have full precision. The maximum error by this method is about 1.5% (rounded up) offset from the real reciprocal. Underflow and overflow are clamped to 0 and infinity as appropriate, and 0 and infinity are given as reciprocals of one another.

4 Shader Processor

Both vertex and fragment shader processors operate using the same architecture. Data is processed as 4-element vectors of floats, referred to as quads. Each processor contains 4 sets of registers for input, constants (shared by all processors of each type), scratch registers, and output. Instructions are 48-bit and allow rearranging operand elements (swizzle) and negation, as well as masking which elements are written to the destination.

4.1 Instruction format

Instruction format is as follows:

```
ooooooooooooooobbbbbbbbbbbbbaaaaaaaaaaaaaddddddd
```

`o`: 15-bit opcode
`b`: 13-bit operand 2
`a`: 13-bit operand 1
`d`: 7-bit destination

4.2 Operand format

Operand format is as follows:

```
rrrrwwzzyyxxn
```

`r`: Source register index
`w`, `z`, `y`, `x`: Index of each swizzle component source; i.e. the operand gets `{r[x],r[y],r[z],r[w]}`.
`n`: High to negate source, low to leave unchanged.

Instruction	Opcode	Clocks	Description
MOV	000000000000000	1	Destination gets operand a value.
MUL	000000000000001	2	Destination gets elementwise product of operands.
ADD	000000000000010	1	Destination gets sum of operands.
DP3	000000000000011	2	All elements of destination get dot product of first three components of operands.
DP4	000000000000100	2	All elements of destination get dot product of operands.
END	000000000000101	1	Indicates end of shader program; no effect on registers.

Table 3: Supported Instructions

4.3 Destination format

Destination format is as follows:

`rrrwxyz`

`r`: Destination register index

`w`, `z`, `y`, `x`: Mask for each component, high to write result and low to leave unchanged.

4.4 Registers

Register	Source index	Destination index	Vertex shader function	Fragment shader function
IN0	0	Cannot be written	Data input	Fragment coordinate (screen space) input
IN1	1			Interpolated vertex attribute 1
IN2	2			Interpolated vertex attribute 2
IN3	3			Interpolated vertex attribute 2
CONST0	4	Cannot be written	Vertex shader constant value	Fragment shader constant value
CONST1	5			
CONST2	6			
CONST3	7			
SCRATCH0	8	0	Vertex shader scratch register	Fragment shader scratch register
SCRATCH1	9	1		
SCRATCH1	10	2		
SCRATCH1	11	3		
OUT0	Cannot be read	4	Vertex shader coordinate output	Fragment color
OUT1		5	Vertex shader attribute output	Unused
OUT2		6		
OUT3		7		

Table 4: Shader processor registers

5 Rasterize Algorithm

Vertex outputs are processed in groups of 3 by the rasterizer. The first output vector is treated as coordinates of the vertex in 4D clip space, and is converted to normalized device coordinates (NDC) as follows for each vertex:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \to \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1/w \end{pmatrix}$$

The x and y coordinates in NDC are then cast to integer pixel coordinates by $[-1.0, 127/128] \to [-128, 127]$, where $(0, 0)$ is the center of the display. Values outside the range are clamped to the endpoints. The other output vectors from each vertex are treated as attributes and also scaled by $1/w$ in preparation for interpolation for each fragment. The area of the triangle is computed using the edge function, defined as a function of three 2-component vectors:

$$ef(v_1, v_2, v_3) = (v_{3,0} - v_{1,0})(v_{2,1} - v_{1,1}) - (v_{3,1} - v_{1,1})(v_{2,0} - v_{1,0})$$

and the area

$$A = ef(v_1, v_2, v_3)$$

where v_1, v_2, v_3 are the x and y components of the NDC coordinates of each vertex. The rasterizer then loops through pixel coordinates across a rectangle spanning between the minimum and maximum pixel x and y of the vertices. At each pixel, a potential fragment is generated, and the pixel coordinates are cast back from integers to floats in NDC using the reverse of the mapping used to cast from float to integer. For each fragment, barycentric coordinates (indicating how close the fragment is to each vertex) are computed as follows, where px is the pixel x and y NDC coordinates:

$$\lambda_0 = \frac{ef(v_2, v_3, px)}{A}$$
$$\lambda_1 = \frac{ef(v_3, v_1, px)}{A}$$
$$\lambda_2 = \frac{ef(v_1, v_2, px)}{A}$$

If all barycentric coordinates are positive, or all are negative, the fragment is inside the triangle (the sign indicating which face); otherwise the fragment is outside the triangle and discarded. The depth and $1/w$ are interpolated using barycentric coordinates:

$$z_{frag} = \lambda_0 v_{1,z} + \lambda_1 v_{2,z} + \lambda_2 v_{3,z}$$
$$\frac{1}{w_{frag}} = \frac{\lambda_0}{v_{1,w}} + \frac{\lambda_1}{v_{2,w}} + \frac{\lambda_2}{v_{3,w}}$$

and perpective-correct interpolation is performed for each the vertex attributes:

$$attr_{frag} = \frac{\lambda_0 v_{1,attr}/v_{1,w} + \lambda_1 v_{2,attr}/v_{2,w} + \lambda_2 v_{3,attr}/v_{3,w}}{w_{frag}}$$

The fragment depth is then compared to the depth stored in the depth buffer at this pixel. If the depth is smaller (the new fragment is in front) it is emitted into the fragment FIFO to be processed by fragment shaders and output to the display, and its depth is written to the depth buffer.

6 Display Output

Frames to be displayed are stored in two framebuffers, one of which is “active”, meaning fragment shaders will write their output to it, and the other of which is “inactive”, meaning its data is unchanging and it will be output to the display. The active and inactive framebuffers can be swapped by a control input, and directly after swap the newly inactive frame will be output to the display. Before the system writes to an active frame, the frame is filled with black pixels to reset it.

The GPU is designed to output to the Adafruit 1.8" 160x128 TFT display breakout board, but it should work with the ST7735R chip with the same resolution display in general, possibly with changes to initialization settings.

7 Application Information

From a user perspective, the process of controlling the GPU is approximately as follows:

1. Reset PLL
2. Reset system
3. Initialization:
   (a) Write vertex count and base address registers
   (b) Write vertex data
   (c) Write vertex program
   (d) Write fragment program
4. Main loop:
   (a) Update constant registers (for example, a transformation matrix for vertices)
   (b) Potentially change vertex data if vertices are dynamic
   (c) Write framebuffer swap control bit
   (d) Write begin render control bit
   (e) Wait for frame period

8 Timing

The system expects a 100 MHz clock input, which is used to drive the on board PLL to generate a 20 MHz clock which the entire system runs on. The clock counts for a render cycle are approximately:

1. 20480 clocks to reset depth and frame buffers
2. Vertex shading, times total vertex count divided by 4 (vertices are processed 4 at a time)
   (a) Vertex data size times 4 elements per vector times 4 vertices for loading inputs
   (b) Total of vertex program instruction clocks
   (c) Vertex data size times 4 elements per vector times 4 vertices for saving outputs
3. Rasterize triangle setup, times total vertex count divided by 3
   (a) Vertex data size times 4 elements per vector times 3 vertices for loading
   (b) 12 clocks for triangle processing
   (c) 2 clocks per fragment if greater than the below item, since the pipeline runs in parallel with fragment shading but can act as a bottleneck
4. Fragment shading, times total fragment count divided by 16 (vertices are processed 4 at a time)
   (a) Vertex data size times 4 elements per vector times 16 fragments for loading inputs
   (b) Total of fragment program instruction clocks

And the clock count for outputting the framebuffer to the display is 20480 pixels times 3 colors per pixel times 8 bits per color times 4 system clocks per SPI clock, giving 1,966,080 clocks.

Also note that the write clock input is delayed by 2 clock cycles for synchronization, and then followed by an edge detector, so the system data in bus requires at least 3 20 MHz periods, or 150 ns, of hold time.

A Module Operation Details

A.1 Shader Processor

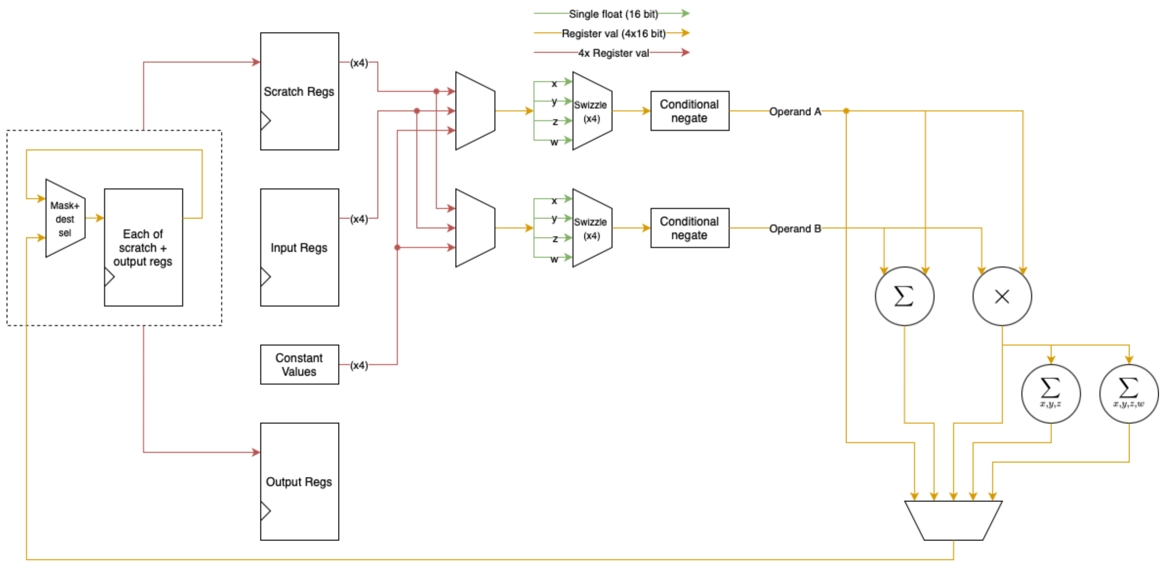

Figure 3: Shader processor block diagram

The individual shader processors contain just registers and arithmetic modules as pictured in Figure 3, with control lines as inputs to the module. Not pictured are instruction enable line which controls any result writing, instruction op state which controls writing of 2-clock instruction (anything involving multiply), and input register write line which allows data input to be stored in input registers, one component at a time.

A.2 Vertex Shader Control

The vertex shader controller operates based on a simple FSM, which iterates through the following states:

1. IDLE - waits for render begin
2. LOAD - loops through each element of each input of each vertex processor and loads from vertex data RAM
3. RUN - runs the vertex shader program, loading the IR with each instruction and executing it for one or two clocks depending on the instruction
4. SAVE - saves each element of each output of each vertex processor into vertex data RAM, then goes back to LOAD if more vertices need shading or IDLE if finished

The vertex shader control module also holds the vertex shader constants and logic to update them from the system bus.

A.3 Rasterizer

The rasterizer has two major components: an FSM to perform the setup required for each triangle, and a pipeline to process each individual fragment in the triangle. The FSM iterates through the following states (more details on the math involved in Section 5):

1. IDLE - waits for rasterize begin or depthbuffer clear begin
2. LOAD_VERTICES - loops through each element of each input from each of 3 triangle vertices and loads from vertex data RAM

3. RECIP_W - takes the reciprocal of w of each vertex in the triangle
4. V1_TO_NDC - multiplies vertex 1 coordinates by 1/w to convert to NDC
5. V2_TO_NDC - multiplies vertex 2 coordinates by 1/w to convert to NDC
6. V3_TO_NDC - multiplies vertex 3 coordinates by 1/w to convert to NDC
7. GET_PX_COORDS - converts floating point NDC coordinates to integer pixel coordinates
8. GET_REC_BOUNDS - find the max and min of pixel x and y coordinates which bound the rectangle around this triangle
9. V1_ATTR_SCALE - scales the attributes of vertex 1 by 1/w
10. V2_ATTR_SCALE - scales the attributes of vertex 2 by 1/w
11. V3_ATTR_SCALE - scales the attributes of vertex 3 by 1/w
12. AREA_COMP - computes the triangle area using the edge function
13. RECIP_AREA - computes the reciprocal of the triangle area
14. GEN_FRAGS - loops through every pixel in the rectangle bounds and emits a fragment into the pipeline for that pixel, then goes back to LOAD_VERTICES if there are more triangles and IDLE if not
15. CLEAR_DEPTHBUFF - loops through every pixel in the depth buffer and sets it to infinity, then goes back to idle

The FSM has 4 associated multipliers and 3 reciprocal calculators, which are used to perform all of the relevant operations on different inputs during the process of the FSM. The pipeline has the following stages, advancing every 2 clocks to account for multiply having a 1 clock delay:

1. Cast pixel coordinates back to floating point
2. Use edge function to find barycentric coordinates
3. Test if fragment inside triangle and normalize barycentric coordinates by triangle area
4. Interpolate z, 1/w using barycentric coordinates
5. Interpolate vertex attributes using barycentric coordinates
6. Scale vertex attributes by interpolated 1/w
7. Test depth against depth buffer, emit fragment into FIFO and update depth buffer if test passed

Each stage in the pipeline has its own dedicated calculation hardware to enable processing fragments at all stages at once. While the pipeline can advance every 2 clocks, it can also pause when the fragment FIFO is full, and will only advance once fragment shaders pop fragments from the FIFO to make room.

A.4 Fragment FIFO

The fragment FIFO is implemented as a circular buffer, each element containing four quads as coordinate and attribute data for one fragment, as well as the pixel coordinates of that fragment. The FIFO also contains a done bit which is set by the rasterizer once all fragments have been output and reset once new data is pushed to the FIFO.

A.5 Fragment Shader Control

The fragment shader controller operates based on a simple FSM very similar to that of the vertex shader controller, which iterates through the following states:

1. IDLE - waits for FIFO not empty which indicates beginning of fragment shading
2. LOAD - waits for FIFO not empty, then loops through each element of each input of each fragment processor and loads from the FIFO head element
3. RUN - runs the fragment shader program, loading the IR with each instruction and executing it for one or two clocks depending on the instruction
4. SAVE - saves color output of each fragment processor into framebuffer, then goes back to LOAD if more fragments need shading or IDLE if finished

The fragment shader control module also holds the fragment shader constants and logic to update them from the system bus.

A.6 Framebuffer and Display Control

The display control module contains both framebuffers and handles writing to the active one and outputting from the inactive one. It also contains an FSM to control sending the initialization sequence and framebuffer data to the display over SPI, which iterates through the following states:

1. SWRESET_SEND - sends software reset command (beginning of init)
2. SWRESET_SPI_WAIT - waits for SPI send to complete
3. SWRESET_DELAY - delays 50 ms after reset
4. SLPOUT_SEND - sends out of sleep mode command
5. SLPOUT_SPI_WAIT - waits for SPI send to complete
6. SLPOUT_DELAY - delays 500 ms after out of sleep
7. INIT_BYTE_SEND - sends byte of initialization sequence (repeats for every byte not requiring a delay)
8. INIT_BYTE_SPI_WAIT - waits for SPI send to complete
9. NORON_SEND - sends normal display on command
10. NORON_SPI_WAIT - waits for SPI send to complete
11. NORON_DELAY - delays 10 ms after display on
12. DISPON_SEND - sends main screen on command
13. DISPON_SPI_WAIT - waits for SPI send to complete
14. DISPON_DELAY - delays 100 ms after screen on
15. IDLE - waits for framebuffer swap
16. RAMWR_SEND - sends RAM write command to begin frame transmission
17. RAMWR_SPI_WAIT - waits for SPI send to complete
18. PX_R_SEND - sends red pixel value
19. PX_R_SPI_WAIT - waits for SPI send to complete
20. PX_G_SEND - sends green pixel value

21. PX_G_SPI_WAIT - waits for SPI send to complete
22. PX_B_SEND - sends blue pixel value
23. PX_B_SPI_WAIT - waits for SPI send to complete, returns to PX_R_SEND to send next pixel or IDLE if all pixels sent

A.7 Top-level Control

The top level module connects the system together and contains a simple FSM for controlling the system, which iterates through the following states:

1. IDLE - waits for write to render start bit in control register
2. RESET_BUFFERS_BEGIN - triggers start of resetting depth and frame buffers
3. RESET_BUFFERS - waits for buffer resetting to finish
4. VERTEX_SHADE_BEGIN - triggers start of vertex shading
5. VERTEX_SHADE - waits for vertex shading to finish
6. RASTERIZE_FRAG_SHADE_BEGIN - triggers start of rasterize and fragment shading
7. RASTERIZE_FRAG_SHADE - waits for fragment shading to finish, then returns to IDLE

B Testing and Supporting Code

B.1 Entire system testing

`GPU_model.py` implements a model of the entire system in python, including a demo with two rotating triangles, and `rayGPU_tb.vhd` runs a simulation of the entire system programmed with the same demo. Neither is instrumented to give a full set of test vectors because of the complexity of the system and differences between the floating point math, but the combination of the two is useful for testing by manually comparing outputs at various stages by adding print statements to the python code and inspecting signal values in the simulator.

B.2 Floating point math testbenches

Both the floating point adder and multiplier modules have testbenches which thoroughly test the module. The testbenches run vectors from a file, generated by `gen_float_tests.py`. The multiplier tests consist of the following edge cases:

- $0 \cdot 0 = 0$
- $0 \cdot n_{min} = 0$
- $n_{max} \cdot 0 = 0$
- $2^{-14} \cdot 2^{-14} = 0$
- $2^{-14} \cdot 2^{-1} = 0$
- $(n <= 2^{-8}) \cdot (n <= 2^{-8}) = 0$
- $\infty \cdot n_{min} = \infty$
- $n_{max} \cdot \infty = \infty$
- $n_{max} \cdot n_{max} = \infty$
- $n_{max} \cdot 1.5 = \infty$

- $(n >= 2^8) \cdot (n >= 2^8) = \infty$

followed by 100 test cases with random operands. The adder tests consist of the following edge cases:

- $0 + 0 = 0$
- $0 - 0 = 0$
- $\infty + n = \infty$ with $n \in \{0, -0, \infty, -\infty, \text{random value}, \text{random value}\}$
- $-\infty + n = -\infty$ with $n \in \{0, -0, \infty, -\infty, \text{random value}, \text{random value}\}$
- $n + \infty = \infty$ with $n \in \{0, -0, \text{random value}, \text{random value}\}$
- $n - \infty = -\infty$ with $n \in \{0, -0, \text{random value}, \text{random value}\}$
- $n - n = 0$ with $n \in \{2^{-14}, -2^{-14}, 1, -1, n_{max}, -n_{max}\}$
- $2^{-14} - (2^{-14} + 2^{-15}) = 0$
- $-2^{-14} + (2^{-14} + 2^{-15}) = 0$
- $2^{-14} - (2^{-14} + 2^{-24}) = 0$
- $-2^{-14} + (2^{-14} + 2^{-24}) = 0$
- $n_{max} + n_{max} = \infty$
- $-n_{max} - n_{max} = -\infty$
- $n_{max} + 2^5 = \infty$
- $-n_{max} - 2^5 = -\infty$
- $2^{15} + 2^{15} = \infty$
- $-2^{15} - 2^{15} = -\infty$

followed also by 100 test cases with random operands.

B.3 Display output test

`display_control_tester.vhd` implements a slightly modified version of the normal display control module, with the framebuffer RAM modules replaced by ROM which loads a memory configuration file generated by `gen_coe.py`. This module can be used to test initializing and writing an image to the display without the rest of the system generating an image. The python file is currently configured to output one pixel wide horizontal lines which alternate between red, green, and blue.

B.4 Triangle demo MCU code

For demoing the system on hardware, `gpu_control.ino` implements an Arduino sketch to be run on an STM32 processor which programs and runs a demo of two rotating triangles on the GPU. This program also provides an example of a shader program which multiplies vertex coordinates by a matrix in constant registers, along with simple interpolated color attributes in vertices.

C Code Structure

The code files included in the project are:

- `display_control_tester.vhd` - Display output test module
- `display_control.vhd` - Framebuffer management and display output module
- `edge_function.vhd` - Edge function calculator
- `float_adder_tb.vhd` - Floating point adder testbench
- `float_adder.vhd` - Floating point adder
- `float_mult_tb.vhd` - Floating point multiplier testbench
- `float_mult.vhd` - Floating point multiplier
- `float_recip.vhd` - Floating point reciprocal calculator
- `float_to_color.vhd` - Cast from floating point to integer 6-bit color module
- `float_to_pixel.vhd` - Cast from floating point to integer 8-bit pixel coordinate module
- `fragment_control.vhd` - Fragment shading control module
- `fragment_FIFO.vhd` - FIFO queue for emitted fragments from rasterizer
- `gen_coe.py` - ROM configuration generator for display output tester
- `gen_float_tests.py` - Floating point math test vector generator
- `gpu_control.ino` - STM32 code to run triangle demo on hardware
- `gpu_float.vhd` - Library with floating point type and helper functions to extract components
- `GPU_model.py` - Python model of full GPU system and triangle demo
- `pixel_to_float.vhd` - Cast from pixel coordinates to floating point
- `rasterizer.vhd` - Rasterizer FSM and pipeline module
- `rayGPU_tb.vhd` - Testbench to simulate entire system with triangle demo
- `rayGPU.vhd` - Top-level GPU module
- `shader_proc.vhd` - Shader processor module
- `TFT_SPI.vhd` - SPI controller for sending data to display
- `vertex_control.vhd` - Vertex shading control module