6.111 FPGA Voxel Ray Tracer

Cece Chu and Parker Huntington

December 13, 2019

1 Introduction

The goal of this project was to create an integer based ray tracer. The basic operating principle of this device is that there exists a scene built up of unit voxels with side lengths of 2^{16}. In order to generate the image corresponding to the camera, a ray is sent out from the camera position through an imaginary pixel of the camera. This ray is propagated through space until it encounters a solid object, which determines its final color.

In order to do this, we needed to create a ray tracer, but also a supporting memory and video system in order to run the ray tracer and display the final image.

2 Ray Tracer (Parker)

2.1 Architecture

The ray tracer can broken down into three modules. The first module is the 'Ray Config' module which interacts with the configuration port of the ray tracer to configure and run the rest of the ray tracer. The operation of the configuration module is described in detail in 2.3. The 'Ray Generator' is run by the configuration module and is responsible for generating the camera rays and calculating which memory address the resultant pixel should be saved to. Finally, these generated camera rays are passed off to one of any number of 'Ray Unit' modules which propagate the ray and calculate their final color.

2.1.1 Ray Generator

The ray generator takes in the directional camera information, frame dimensions, and frame address from the configuration module, then generates one ray direction per pixel along with the pixel address. The pixels are generated from left-to-right then top-to-bottom, the same order as the pixels are displayed on the screen. The resultant ray direction may be normalized by the ray generator if the 'normalize' signal is set. The normalization is performed using a newton's method approach using two passes (Fig. 2).

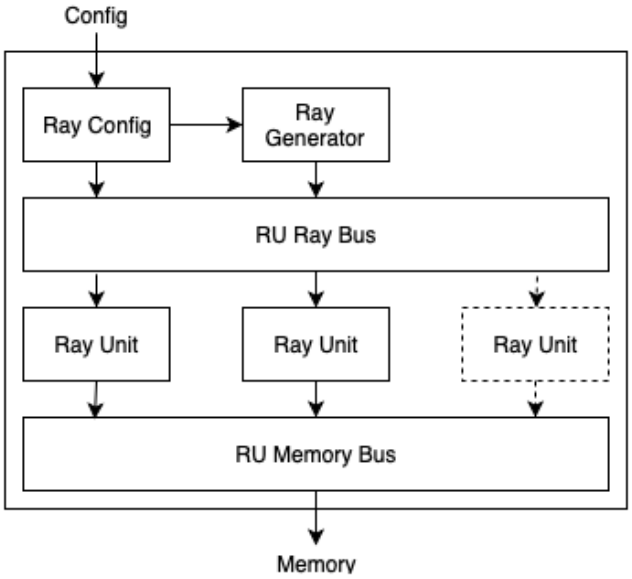


Figure 1: The configuration port drives the 'Ray Config' module which controls the rest of the system. The 'Ray Generator' feeds rays to the 'Ray Unit's which interact with the rest of the system through the memory bus.

```
def newtons_method(x, y, v):
    norm = (x**2 + y**2 + v**2) >> 16;

    # generate an initial guess
    bits = 16
    toShift = bits - int(log(norm, 2))
    y = 1 << bits + (toShift // 2)

    for newton_step in range(2):
        # three in fixed point
        three = 3 << bits

        # convert y back to f16
        y = y * (three - ((n*y)*y)) >> bits + 1

    # 15//16 safety margin
    return y * 15//16
```

Figure 2: Python description of the normalization method used by the ray generator.

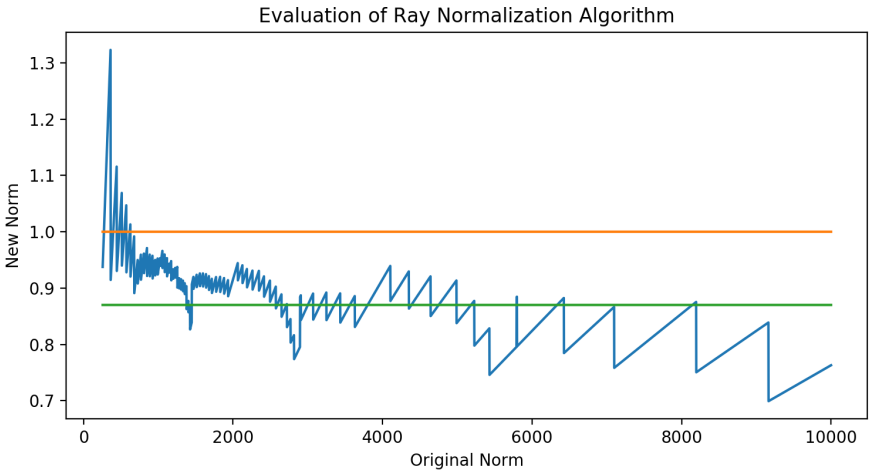


Figure 3: Ideally, the ray normalization would result in the ray being 15/16 in order to keep the norm small enough to prevent integer overflow.

The normalization algorithm helps to ensure that small vectors are scaled up to improve performance and ensure that the ray propagation can proceed correctly.

While ideally, the normalization algorithm would increase the maximum field of view, it performs best with smaller vectors as can be seen in Fig. 3. Unfortunately, using small vectors means that the angular resolution for camera rotations is decreased. Thus, under most cases it is recommended that a moderate FOV is used with prescaled camera vectors and no normalization (Section 2.3.3).

2.1.2 Ray Unit

The ray unit is the basic unit responsible for taking a camera ray, propagating it, then calculating the final color that corresponds to the rays pixel. This process can be broken up into three steps. First, the ray unit takes the initial ray position and looks up the leaf node properties corresponding to that position. This gives the leaf node bounding box as well as the material information. If the material is empty, then the ray is propagated until it is just outside of the bounding box. After this, the ray unit returns back to step one. If the material isn't empty, then the ray unit writes the final pixel value back to memory.

In order to make this process easier, the ray unit contains two submodules. The 'RayMemory' submodule is used to perform the material lookup as well as the pixel write operation, while the 'RayStepper' submodule is used to propagate the ray. The ray unit then contains a top level state machine that is used to control the other two modules.

2.1.3 Ray Memory

The ray memory module performs material lookups and pixel write operations. In order to look up a material. The ray memory module queries the root node of the octree. Using the MSB of the position vector, the correct octant is selected, and looked up in the tree. This process is repeated until a leaf node is found. If the leaf node represents material 0, then the ray memory module indicates that the position is empty, otherwise it looks up the material and returns it.

Additionally, the ray unit may look up the background material that is used for out of bounds rays, which is simply the first item in the material table.

2.1.4 Ray Stepper

The ray stepper is responsible for propagating the ray through space. In order to do this a simple binary search algorithm is used. The initial position is loaded into an accumulator vector while the direction is scaled up so that it is at least longer than the maximum length of the bounding box. Next the accumulator position is added to the direction vector. If the resulting value is within the boundary box, then the sum is committed to the accumulator. If the position is one unit outside of the bounding box, then that position is returned. Otherwise, the change isn't committed to the accumulator register.

As long as the ray tracer hasn't returned it will repeat the previous steps every cycle while bisecting the direction vector each time. An important problem to realise with this algorithm, however, is that it isn't guaranteed to converge when working with positive vector components. This is due to the fact that when the vector is bisected, it is rounded down and the binary search is no longer guaranteed to converge. In order to prevent this form happening, an extra bit is stored in the direction vector to represent the half digit. If the direction component is positive, then this will be used to round up the vector during the computation.

2.2 Connections and Usage

The top level ray tracer module is fairly simple in terms of the ports that it exposes. The 'clock' and 'reset' ports of provide the clock input and an active high reset for the entire ray tracer. The interrupt port signals that the ray tracer is no longer busy. Note, however, that the ray tracer can be ready to render a new frame earlier. It is possible to have multiple frames being rendered at the same time in the ray tracer, however, it is currently difficult to distinguish when a frame is finished, so this usage is discouraged.

The ray tracer also exposes two memory ports, a slave port and a master port. The slave port is used to access the configuration registers inside of the ray tracer. The memory configuration base address is set when instantiating the ray tracer module through the 'CONFIG_ADDRESS' parameter and occupies 32 address locations in the slave address space. The master port is used by the ray tracer to both request scene information and to set the pixels in the frame

Offset	Value
0x00	Status
0x01	Material Address
0x02	Tree Address
0x03	Frame Address
0x04	Camera $\hat{q}$
0x07	Camera $\hat{v}$
0x0A	Camera $\hat{x}$
0x0D	Camera $\hat{y}$
0x10	Frame width
0x11	Frame height

Table 1: The offset of the base address is given for each of the configuration values.

buffer. The ray tracer occupies id in the master is space. The base master id is set by the 'MASTER_ID_BASE' parameter and spans however many ray units that are inside of the ray tracer. The ray tracer master doesn't need access to the slave port, and as such they can be on separate buses.

2.3 Configuration

The status register is used to both set/ready the control signals and get the ray tracer status.

2.3.1 Configuration Register (0x00)

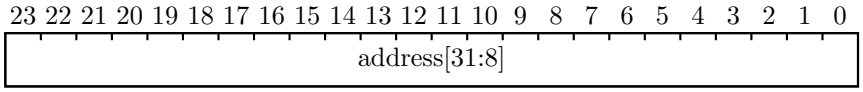
The start field 'S' is a write only field that is used to start the ray tracer by writing a 1 to it. The ready field 'R' is a read only field that indicates whether or not the ray tracer is ready to start a new frame, but doesn't indicate that the last frame is finished. The busy signal 'B' indicates whether or not the ray tracer is currently working on a frame.

The flush field 'F' is a write only field that is used to flush any cached scene information from the ray tracer. This field should be asserted after any changes to the scene information that the ray tracer is pointing to ensure that all of the data will be up to date during the next render.

The reset field 'R' is used to reset the entire ray tracer. This should be performed in order to do a soft reset on the system. Care should be taken to flush the memory bus afterwards, especially if the ray tracer was operating, to ensure any in route requests are flushed from the system and don't cause the memory bus to deadlock.

The normalization control field 'N' is a read/write field used to turn on and off the camera vector normalization. It is recommended that this be left off.

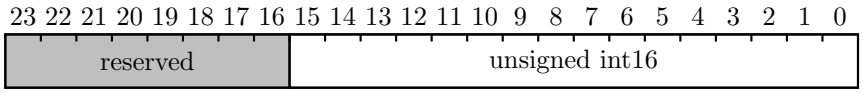
2.3.2 Address Registers (0x01 - 0x03)



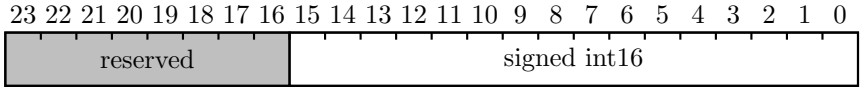
As the address size of the ray tracer is 32 bits while the data is only 24 bits, the address configuration registers only specify the upper 24 bits of the address while the lower 8 bits are set to 0. Thus the frame buffer, material table, and octree must be aligned to aligned to the nearest 1024 in the PS address space.

2.3.3 Vector Registers (0x04 - 0x0F)

Each vector quantity in the ray tracer is specified by a set of three 16-bit vector registers. Each set of three registers is organized so that the x-coordinate is specified first (lowest offset), then the y, and z components. All of the vector quantities in the ray tracer configuration relate to the camera description: position ($\hat{q}$), direction ($\hat{v}$), x-axis ($\hat{x}$), and y-axis ($\hat{y}$).



The camera position is an unsigned vector, and is used to describe the position of the camera in the scene, where the camera position can be any 16-bit 3-vector.



The other camera quantities are signed vectors. The $\hat{v}$ quantity gives the normal vector to the camera plane, or the direction that the camera is pointing. The $\hat{x}$ value is the vector that describes how far to change the camera direction in order to move right 1 pixel on the x-axis. The $\hat{y}$ value similarly describes the direction to move in order to move up one pixel on the y-axis. While the $\hat{x}$, $\hat{y}$, and $\hat{v}$ values should be orthogonal in order to describe a 'normal' camera they can be changed to skew the camera image.

The $\hat{x}$ and $\hat{y}$ values should have the same magnitude in order to ensure that the final image isn't stretched in one direction. The field of view of the camera can be adjusted by changing the relative $\hat{x}, \hat{y}$ to $\hat{v}$ magnitude. In order to prevent overflow inside the ray tracer while it is generating the rays, (1) should be satisfied. In order to retain angular resolution while rotation the camera, (1) should be as close to saturating as possible, otherwise the $\hat{x}$ and $\hat{y}$ vectors will

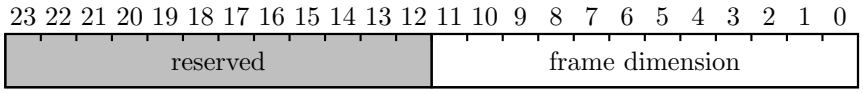
be so small, that there isn't enough precision to rotate them without causing the camera to skew.

$$\left|\hat{v} + \frac{width}{2}\hat{x} + \frac{height}{2}\hat{y}\right| < 2^{15} - 1 \tag{1}$$

Due to optimizations in the ray stepper that are discussed elsewhere, the ray must also meet the minimum size bound given by (2) where n is the minimum depth of the octree being rendered.

$$|\hat{v}| > 2^{15-n} \tag{2}$$

2.3.4 Frame Dimension Registers (0x10 - 0x11)



The dimensions of the frame buffer are described by the read/write frame dimension registers which hold the dimension as an unsigned 12-bit value.

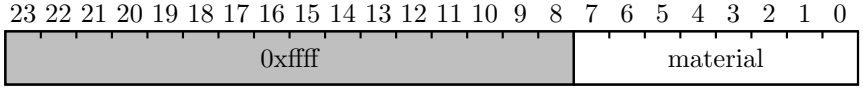
2.4 File Format

The ray tracer scene information is split into two sections. The tree section is used to divide up the space in the scene into regions which are labeled with a material. The material table is then used to look up the properties of that region of space.

The tree is given as a list of nodes where each node represents a subdivision in space. The node contains 8 24-bit entries where the offset of the entry gives the octant that it specifies. This offset is given by

$$\text{offset} = z * 4 + y * 2 + x, \tag{3}$$

where x, y, and z are either 0 or 1. Each entry can either specify another node, or a material. An entry is a material when the upper 16 bits are all 1, as seen below.

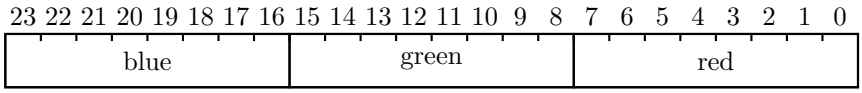


In the special case that the material is 0, the space is considered to be empty. Otherwise, material is the index used to look up the material in the material table. If the entry isn't a material, then it represents another node and is interpreted as a 24-bit number giving the index of the index of next node (Fig. 4). The root node of the tree is always taken to be the node at index 0. The tree should always form an acyclic graph so that infinite loops aren't encountered.

```
# root node
0xffff_01
0xffff_01
0xffff_01
0xffff_01
1
0xffff_00
0xffff_00
0xffff_00
# node 1...
```

Figure 4: Example root node. All of the octants on the bottom (z-axis) are set to material 1, while on top everything is empty space except for the lowest $x, y = 0$ octant which is described by node 1.

The material table is a 256-element list of 24-bit material entries. The material at index 0 is a special case that represents the background used when a ray is detected to have gone out of bounds. each material entry specifies the color of the material as seen below.



2.5 Implementation Process

In order to implement and test the ray tracer, the system was slowly built up in a comprehensive simulation environment. First I would develop the specification for a new systemverilog module, then add a verilator command in order to compile it and check for syntax errors. The verilated output was then wrapped in a c++ class that was used to add convenience functions so that the system wouldn't be hard to use by other code. This wrapper could then be called by test cases, and sometimes used by a routine to simulate the final image output of the ray tracer. This proved to be a very effective development cycle for the ray tracer and allowed me to do a lot of development on the ray tracer before the memory system was ready. Additionally, I could quickly test the code without putting on the hardware and was able to find many bugs, so much so that the ray tracer presented no problems when put onto the actual fpga (besides critical path issues that were fixed by lowering the clock speed).

In the future I certainly want to build on this development style. I would, however, want to experiment around with some alternatives to systemverilog to see if there is anything that produces cleaner code.

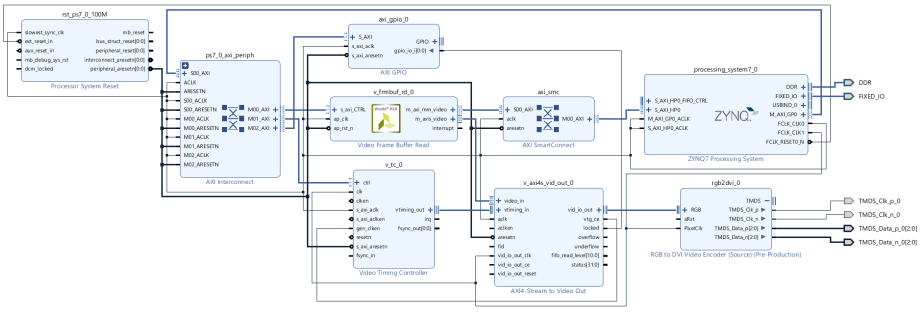


Figure 5: Block Diagram of Display Pipeline.

3 Display system (Cece)

3.1 Overview/Motivation

The goal of the display subsystem was to be able to display the frame buffer on an external monitor. While the task of displaying an image is quite similar to what we did in the labs during the semester, the task at hand for our final project was different in some significant aspects. First, there is the difference in pixel generation. Whereas in the labs we generated pixels either combinationally or from BRAM, in this project due to the size and resolution of the image we needed to read out pixels from the frame buffer in DRAM. Second, there is the difference in video format. The Pynq board we used does not have VGA output, instead it only supports HDMI.

The tricky part of pixel generation was the need to be able to sequentially read out pixel values from DRAM fast enough to meet the speed requirements at our given resolution. For example, in order to display a 720p (1080 x 720 pixels), 8-bit RGB (24 bits per pixel) image at 30fps, we would need to supply a minimum of 30 * 1080 * 720 * 24 = 560 Mbits per second. The high data rate is the reason why we chose to use the Pynq Z2 board instead of the Nexys 4 DDR, since it has a faster DRAM. In order to actually achieve this rate, we read out the pixels via the built in HP AXI ports on the Zynq processing cores that come with the Pynq system and provide access to the builtin memory controller. The various IPs that made up the display pipeline were all from the IP catalog, so I will not write about the internal implementations but the function and configuration of each is briefly outlined below and connectivity is shown in the diagram.

3.2 Hardware Components

3.2.1 Video Frame Buffer Read IP

The Video Frame Buffer Read IP is essentially serves as a Direct Memory Access (DMA) module. It requests data from DRAM on the AXI interface and outputs

pixel data as an AXI stream. Since we didn't want to have to deal with offset addresses, each pixel took up 32 bits instead of 24, aka RGBX8 format where one byte is empty. Accordingly, we had to customize the IP to support a RGBX8 data format. Frame size and data format were configured in software.

3.2.2 Video Timing Controller IP

The Video Timing Controller (VTC) IP provides VGA timing signals (hsync, vsync, blank) given an input clock. The default frame size is set in the hardware but frame size was also configurable via software. The VTC supports both timing generation and detection (if given an input video stream), but for our purposes we just used the generation capability.

3.2.3 AXI Stream to Video Out IP

The AXI Stream to Video Out IP takes in the AXI Stream pixels from the Video Frame Buffer Read and the timing signals from the VTC and outputs valid VGA signal.

3.2.4 RGB to DVI IP

The RGB to DVI IP converts the VGA video signal from the AXIS-Video Out IP into HDMI. Its outputs were connected to the HDMI ports specified in the XDC file. This IP did not come with the Vivado WebPack and had to be downloaded from the Vivado Master Library.

3.2.5 Clock Domain

All of these IPs on the display pipeline were the same clock domain that is separate from the clock domain of the ray tracing logic. This clock was provided by a second PL fabric clock generated in the Zynq processing system and the frequency was configurable in software to adjust the frame rate for different frame sizes.

3.3 Software Configuration

Since one of our stretch goals was to be able to support different resolutions dynamically (i.e. without having to regenerate the bitstream), this involved having to be able to configure the VTC and Video Frame Buffer Read in software. Both of the IPs were connected as memory-mapped slaves to the Processing System, so we could configure to get different resolutions by writing to the appropriate control registers in the Jupyter Notebooks. A table of different resolutions and the values that had to be written to each register is below.

3.4 Implementation Process/Challenges

The display pipeline was developed entirely separately from the ray tracer and memory subsystems. The goal for the display subsystem was essentially to be able to set a frame buffer address and resolution in the Jupyter Notebook and get a valid HDMI output. The idea was that to the display, it would not make any difference if the pixels were written into the frame via the ray tracer or in python using the PS, so we would be able to seamlessly integrate the two systems together once both were working.

I started by testing the display pipeline present in the Pynq base overlay according to the documentation. This essentially verified that we would be able to get sufficient data bandwidth from the DRAM. However, the block diagram for the base overlay was extremely complex and had many more components than was necessary for our application, therefore we would need to recreate our own display pipeline. The next step was to create a simplified version of the pipeline described above except to replace the Video Frame Buffer Read module with a Test Pattern Generator. The reason for doing this was that the Test Pattern Generator was a much simpler and standalone IP than the Video Frame Buffer Read, not requiring any special software configuration and not needing to interact with DRAM. Thus, I used this pipeline to verify the functionality of the downstream modules – the AXI Stream to Video Out and the RGB to DVI, as well as the topology of the hardware.

Getting the pipeline functional was quite challenging because it was quite opaque and difficult to debug the software settings. I used an AXI GPIO module to read the "locked" signal from the AXI Stream to Video Out module to help indicate whether it was able to generate a valid signal. I then had to read through a lot of documentation and Xilinx-provided C driver code to figure out the essential registers to set, i.e. generate the table above. I first just got a single resolution working by configuring the Video Frame Buffer Read IP and setting the VTC resolution in Vivado as a default, and once that worked I then went on to figuring out the configurations for the VTC. The specific numbers for hsync, vsync and other registers I got from the "Customize IP" window in Vivado. This whole process ended up taking many hours because I was really only able to see the individual status registers and the locked signal, so whenever something didn't work I had to go back to the documentation and example drivers and figure out what was missing. Ultimately, for example, I realized that I had set all the correct registers for changing the VTC frame size but needed to set a certain bit in the control register in order to update the settings. This process ended up teaching me how to read through dense documentation and driver code and extract the necessary information.

4 Memory System (Cece)

4.1 Overview/Motivation

The overarching goal of the memory system is to handle read and write requests to memory. In our system, the masters that can initiate reads and writes are the Ray Tracer and the Computer Interface, which is controlled by the Zynq PS. The slaves that are the memory elements that service the requests are the DRAM, BRAM, and the Ray Tracer Configuration Register. The tasks of the various modules can be broken down into arbitrating between different requests to avoid bus contention, routing requests to the appropriate slave and doing address space conversion, and interfacing with AXI protocol. Arbitration and address space conversion are necessary since we have multiple masters and slaves; interfacing with AXI is necessary since the BRAM and DRAM both use AXI interfaces, however for our custom modules we chose to use a simpler protocol described below.

4.2 Memory Access Protocol

Our internal protocol was sort of like a simpler version of AXI. We had separate channels for Master-to-Slave (MS, for requests) and Slave-to-Master (SM, for read data). Each had the signals Valid, Taken, Master ID (6 bits), and Data (24 bits), and the MS channel had additional Write and Address (32 bits) signals. Master ID specified which master was making the request for the MS channel and the original requester of the read request for the SM channel. Data was either the data to be written for the MS channel or the returned data for the SM channel. The Valid is set high to signal a valid request on the MS channel or valid read data on the SM channel, and Taken is set high to signal that the request or data was accepted by the receiver. A transaction is considered completed when both Valid and Taken are high. A slave takes any valid request it receives, but a master only takes read data when the Master ID matches its own. Accordingly, MS requests are routed to a single destination, but SM data can be broadcast to all the masters on the bus.

4.3 Hardware Components

4.3.1 Crossbar Module

The crossbar is responsible for arbitrating between the two masters on the MS channel and between the three slaves on the SM channel. It also routes requests from the MS channel to the correct slave based on the address. Our address space was divided up as follows: addresses 0x0-0x100 were reserved for the Ray Tracer Configuration Register, addresses 0x100-0x800 were reserved for the BRAM, and addresses greater than 0x800 were reserved for DRAM. After checking the addresses incoming from the master to see which slave the request should be routed to, the base address was subtracted from the incoming address

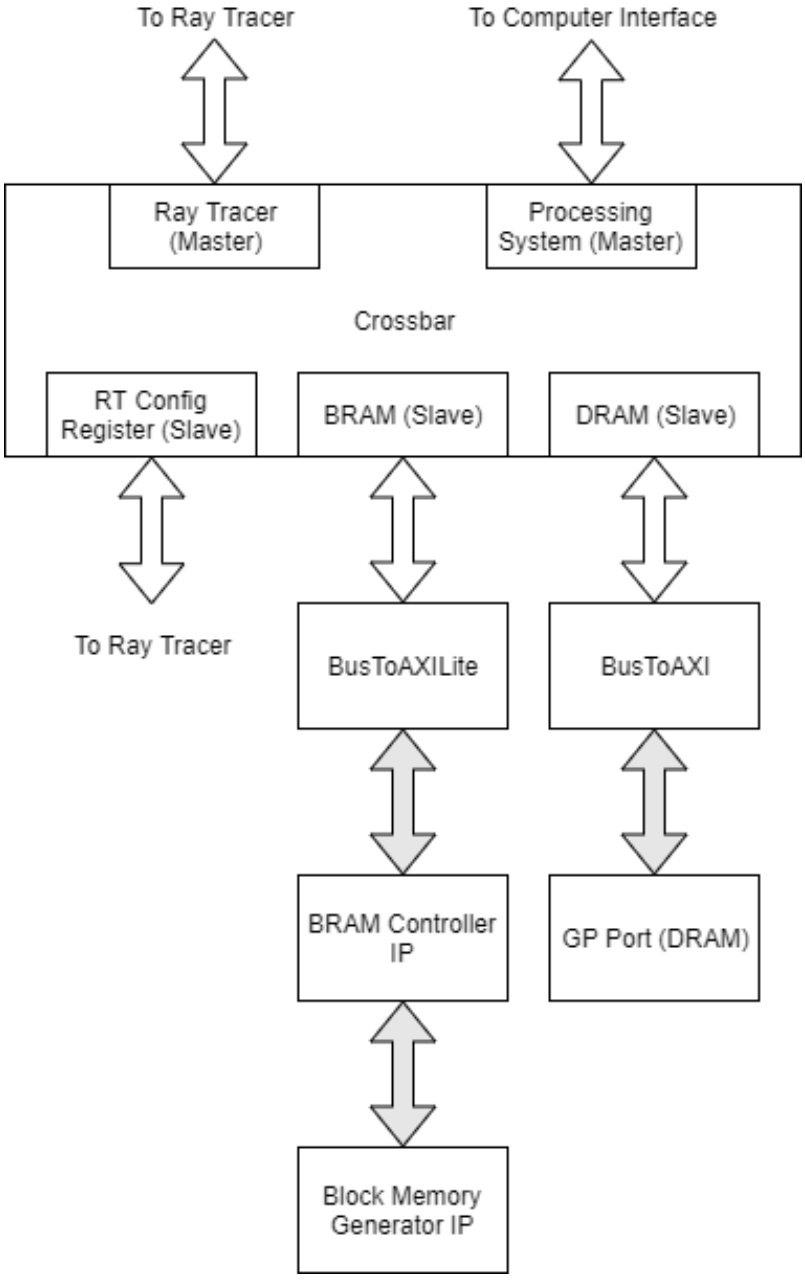


Figure 6: Block Diagram showing architecture of memory system. White arrows represent internal Valid/Taken protocol, gray arrows represent AXI or AXILite protocol.

MRU	1st Priority	2nd Priority	3rd Priority
A	B	C	A
B	C	A	B
C	A	B	C

Table 2: Round Robin style arbitration order given MRU.

before being passed to the slave. For example, address 0x104 from a master corresponds to address 0x4 in the BRAM.

Arbitration for the MS channel was done using a Least Recently Used (LRU) scheme on a per-slave basis. If there was only one master trying to request for any slave at a given time (meaning that its valid signal was high and the address was within that slave's address space), that request would simply be passed to the slave. In the case of contention, i.e. both masters trying to read or write from the same slave, the arbitration logic grants priority to the Least Recently Used (LRU) master for that slave. The LRU state is maintained in a sequential block with a register such that whenever a transaction is completed by one master, i.e. Valid and Taken are both high on the MS channel, the LRU is set to the other master. Apart from the LRU logic, all other arbitration and routing logic is combinational so as not to introduce any cycles of latency.

Arbitration on the SM channel is done using a round robin scheme. The Most Recently Used (MRU) slave is the slave that has most recently completed a transaction, i.e. had valid and taken both high. At any given point, the priority in an arbitration scenario is granted based on the table below, where A = Ray Configuration Port, B = BRAM, and C = DRAM. Like the MS channel, the MRU logic is sequential and the routing logic is combinational so as not to introduce latency.

4.3.2 Binary Arbiter Module

The Binary Arbitrator was sort of a simplified version of the crossbar module. On the MS channel, it provided arbitration between two masters for accessing a single slave port using the LRU scheme described for the crossbar's MS channel. On the SM channel, it simply passed from the slave port to both master ports. It was used internally in the Ray Tracer for arbitrating between different Ray Units' memory requests in a tree structure.

4.3.3 BusToAXI Module

The BusToAXI module served as a "translator" between our internal Ready/Taken protocol and the AXI protocol used by the DRAM. In the system diagram, it resided between the internal memory bus on the crossbar and the GP AXI port on the Zynq PS. The challenge of this module was to figure out how to set the more than 30 signals on the AXI interface to be able to properly send requests and receive DRAM data.

Luckily, our protocol was fairly similar to AXI in principal, since AXI uses ready/valid signals on each of its channels. Since AXI has separate read and write channels, we routed our msValid/msTaken signals to the read or write valid/ready signals depending on the value of our msWrite signal. Since we used a 24 bit data bus but the DRAM AXI interface had a 32 bit data but, we simply padded the upper byte with zeros.

Likewise, the msAddress was simply passed to both the read and write addresses. The DRAM AXI interface also had the nice property of supporting IDs, which were functionally the same as our Master IDs, so we were able to pass msID to wid/arid (write ID and read address ID) and rid (read ID) to smID. Given the assignments of all those basic signals, there were still many other peripheral signals used for supporting bursting, caching, and other advanced AXI functionalities which we did not need but still needed to set properly. The full assignment of values, generated after reading the AXI specification, the Zynq DRAM Controller datasheet, and some trial and error, is listed below. See also the verilog for thie module in the Appendix.

4.3.4 BusToAXILite Module

The BusToAXILite module was functionally very similar to the BusToAXI module except that it served as the interface between our internal protocol and the BRAM Controller, which had the option of using AXI Lite. AXI Lite does not support bursting or caching, so there are far fewer signals to deal with. The core signals are translated in the same way as above, the primary difference being that IDs are not supported in AXILite. However, since the BRAM has a read latency of 1 cycle, we were able to support master IDs simply by writing the msID into a register whenever there was a read request and wiring smID to the contents of that register. The full table of AXILite signals is summarized below.

4.3.5 BRAM Controller and Block Memory Generator IPs

These two IPs actually generated the block memory and created the AXILite interface for it. The block diagram showing the connections of these IPs is in the figure below. Since we had the Block Memory Generator connected to the controller, we were able to set the depth of the BRAM using Vivado's Address Editor.

4.4 Implementation Process/Challenges

I began the implementation process by writing the Binary Arbitrator module in SystemVerilog since it was a good way to make sure I understood how our internal protocol worked and also relatively easy to verify with a simulation test bench. I then worked on the BusToAXILite module to interface with the BRAM. Due to the relatively small number of extraneous AXILite signals, this was relatively easy to get working in simulation, and I was able to read and

Signal	Direction	BusToAXI Assignment	AXILite?
aready	SM	msTaken = write ? awready && wready : aready	Y
awready	SM	msTaken = write ? awready && wready : aready	Y
bvalid	SM	ignored	Y
rlast	SM	ignored	N
rvalid	SM	smValid	Y
wready	SM	msTaken = write ? awready && wready : aready	Y
bresp	SM	ignored	Y
rresp	SM	ignored	Y
bid	SM	ignored	N
rid	SM	smID	N
rdata	SM	smData	Y
arvalid	MS	!write && msValid	Y
awvalid	MS	write && msValid	Y
bready	MS	1	Y
rready	MS	smTaken	Y
wlast	MS	write && msValid	N
wvalid	MS	write && msValid	Y
arburst	MS	0	N
arlock	MS	0	N
arsize	MS	3'b010	N
awburst	MS	2'b01	N
awlock	MS	0	N
awsize	MS	3'b010	N
arprot	MS	2'b01	Y
awprot	MS	0	Y
araddr	MS	msAddress	Y
awaddr	MS	msAddress	Y
arcache	MS	0	N
arlen	MS	0	N
arqos	MS	0	N
awcache	MS	0	N
awlen	MS	0	N
awqos	MS	0	N
arid	MS	msID	N
awid	MS	msID	N
wid	MS	msID	N
wdata	MS	msData	Y
wstrb	MS	4'b1111	Y

Table 3: AXI and AXILite signals.

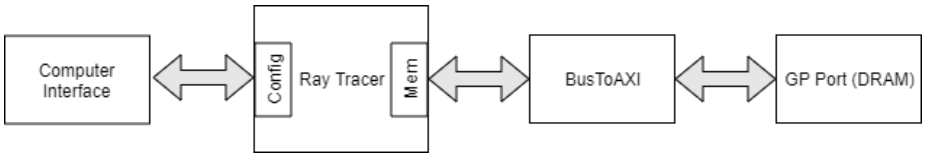


Figure 7: Simplified memory system with only DRAM.

write data. This first iteration of the module, however, only had the useful ready, valid, address, data, etc signals broken out and each wire was connected manually in the block diagram. After the full BusToAXI module was written, I ended up rewriting the BusToAXILite module to include all of the AXILite signals, even the unused outputs, so that Vivado recognized and bundled together the wires as an interface on the block diagram.

Getting the BusToAXI module to work was quite challenging, especially since we were unable to simulate DRAM. After separate failed attempts, Parker and I sat down together and wrote an initial version of the module and tested it using the Pynq framework with the Memory Master Computer Interface. We were able to get it to work but discovered some strange behavior with lagging read requests and only being able to write to 64 bits at a time rather than 32 bits. These issues were eventually fixed with adjustments to the Memory Master state machine itself and by using the GP instead of the HP ports, which have slightly different protocols.

For integrating the various components, we started with a simplified memory architecture where the frame buffer, octree, and materials table were all stored in DRAM. This eliminated the need for a Crossbar, since the Computer Interface could be connected directly to the Ray Tracer Configuration Port, and the Ray Tracer could directly interface with the DRAM via the BusToAXI module. (see the diagram below.) Using this structure, we were able to verify the functionality of the Ray Tracer with the Display Pipeline. At this point, we had a functional, though simplified system.

I then moved to integration of the full system with the BRAM. The goal of this was to move some of the data accesses to BRAM since BRAM is faster than DRAM. Our original intent was to have both the materials table and octree stored in BRAM, however it turned out that the maximum amount of BRAM available on the FPGA, 128K, was not sufficient to hold the octree for larger models, so we just stored the materials table in BRAM. This also meant we could decrease the size of the BRAM – the final bitstream we used had 1024 bytes but the table was only 256 bytes so we theoretically could have shrunk the BRAM down to 256 bytes. On the software side, we wrote the material table into BRAM through the memory master and crossbar and set the material table address in the Ray Tracer Configurations to be the base address of the BRAM (0x100). This resulted in a speed improvement of about 20% compared to having everything in the DRAM.

5 Computer Interface (Parker)

In order to configure our system and run it, we needed an interface that could be used by the computer in order to access our custom memory bus as well as driver software to automate it.

5.1 Memory Master

The memory master module allows arbitrary read and write commands on our custom memory bus from the computer. This allows us to configure the ray tracer and the BRAM, or even inspect and clear out the bus in the event of an error. The module provides a standard memory bus master interface and is controlled by a 32 bit input port, and reads information back via a 32 bit output port. The input is split up, with the upper 8 bits representing a command value, and the lower eight representing a field value for each command.

The memory master essentially works by allowing the computer to configure internal registers setting the master ID, as well as all of the parameters related to any memory request. Once a memory request has been formed, the request can be queued to send. Once a request is queued, the memory master will indicate that there is a pending request until the bus has sent the request. Responses to the memory module are handled by taking the response and storing it into registers that can be read from later. A response will only be received if the master ID matches the one configured by the user. A user can, however, sweep the master ID parameter in order to catch any responses on the bus corresponding to master IDs in that range. Once a response has been read, another can not be read until the read data is cleared.

The easiest way to use the memory master module is to connect its 32 bit input and output ports to an axi gpio channel's input and output ports.

6 Appendix

6.1 BRAM_Wrapper_bd.sv

```
module BRAM_Wrapper_bd #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24,
    parameter BRAM_ADDRESS_WIDTH=15,
    parameter BRAM_DATA_WIDTH=32
    )(
    input clk,
    input rst,
    MemoryBus.Slave bus,

    output logic[BRAM_ADDRESS_WIDTH-1:0] awaddr,
    output logic awvalid,
```

```
    input awready,
    output logic [BRAM_DATA_WIDTH-1:0] wdata,
    output logic wvalid,
    input wready,
    output logic[BRAM_ADDRESS_WIDTH-1:0] araddr,
    output logic arvalid,
    input arready,
    input [BRAM_DATA_WIDTH-1:0] rdata,
    input rvalid,
    output logic rready,
    output logic [2:0] awprot,
    output logic [2:0] arprot,
    output logic bready,
    output logic [3:0] wstrb,
    output logic rstb
    );

    logic[MASTER_ID_WIDTH-1:0] id_fifo[3:0];
    logic[1:0] fifo_count;
    logic msValid_prev;
    logic rvalid_prev;
//    bram_wrapper bram (
//    .S_AXI_0_araddr(araddr),
//    .S_AXI_0_arprot(arprot),
//    .S_AXI_0_arready(arready),
//    .S_AXI_0_arvalid(arvalid),
//    .S_AXI_0_awaddr(awaddr),
//    .S_AXI_0_awprot(awprot),
//    .S_AXI_0_awready(awready),
//    .S_AXI_0_awvalid(awvalid),
//    .S_AXI_0_bready(bready),
//    .S_AXI_0_bresp(),
//    .S_AXI_0_bvalid(),
//    .S_AXI_0_rdata(rdata),
//    .S_AXI_0_rready(rready),
//    .S_AXI_0_rresp(),
//    .S_AXI_0_rvalid(rvalid),
//    .S_AXI_0_wdata(wdata),
//    .S_AXI_0_wready(wready),
//    .S_AXI_0_wstrb(wstrb),
//    .S_AXI_0_wvalid(wvalid),
//    .rsta_busy_0(rsta_busy),
//    .s_axi_aclk_0(clk),
//    .s_axi_aresetn_0(!rst));

    assign rstb = !rst;
```

```
assign arprot = 0;
assign wstrb = 4'b1111;
assign awprot = 0;
assign bready = 1;

assign awvalid = bus.msValid && bus.msWrite;
assign wvalid = bus.msValid && bus.msWrite;
assign awaddr = {bus.msAddress[BRAM_ADDRESS_WIDTH-3:0], 2'b0};
assign wdata = {8'b0, bus.msData};

assign arvalid = bus.msValid && (!bus.msWrite);
assign araddr = {bus.msAddress[BRAM_ADDRESS_WIDTH-3:0], 2'b0};
assign rready = bus.smTaken;

assign bus.msTaken = bus.msWrite ? wready: aready;
assign bus.smValid = rvalid;
assign bus.smData = rdata[23:0];
assign bus.smID = id_fifo[0];


logic pushFifo;
logic popFifo;
assign pushFifo = bus.msValid && !bus.msWrite && bus.msTaken;
assign popFifo = rvalid && bus.smTaken;

always_ff @ (posedge clk) begin
    rvalid_prev <= rvalid;
    msValid_prev <= bus.msValid;
    if (rst) begin
        fifo_count <= 0;
        rvalid_prev <= 0;
        msValid_prev <= 0;
    end else begin
        if (pushFifo) begin
            if (popFifo) begin
                bus.smID <= id_fifo[0];
                for(integer i = 0; i < fifo_count; i = i+1) begin
                    id_fifo[i] <= id_fifo[i+1];
                end
                id_fifo[fifo_count] <= bus.msID;
            end else begin
                id_fifo[fifo_count] <= bus.msID;
                fifo_count <= fifo_count + 1;
            end
        end else begin
```

```
                if (popFifo) begin
                    id_fifo[0] <= id_fifo[1];
                    id_fifo[1] <= id_fifo[2];
                    id_fifo[2] <= id_fifo[3];
                    fifo_count <= fifo_count - 1;
                end
            end
        end
    end
endmodule
```

6.2 BinaryArbiter.sv

```
module BinaryArbiter #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24
    )(
    input clk,
    MemoryBus.Slave sbus0,
    MemoryBus.Slave sbus1,
    MemoryBus.Master mbus
    );
    BinaryArbiter_MS ms(.clk(clk), .sbus0(sbus0), .sbus1(sbus1), .mbus(mbus));
    BinaryArbiter_SM sm(.sbus0(sbus0), .sbus1(sbus1), .mbus(mbus));
endmodule
/* verilator lint_off DECLFILENAME */
module BinaryArbiter_MS #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24
    )(
    input clk,
    MemoryBus.Slave sbus0,
    MemoryBus.Slave sbus1,
    MemoryBus.Master mbus
    );
    logic lru;
    logic select;

    always_comb begin
        select = sbus1.msValid && !(lru && sbus0.msValid);

        if (select) begin
            //pass from sbus0 to mbus
            mbus.msID = sbus0.msID;
```

```
            mbus.msAddress = sbus0.msAddress;
            mbus.msData = sbus0.msData;
            mbus.msWrite = sbus0.msWrite;
            mbus.msValid = sbus0.msValid;
            sbus0.msTaken = mbus.msTaken;
            sbus1.msTaken = 0;
        end else begin
            //pass from sbus1 to mbus
            mbus.msID = sbus1.msID;
            mbus.msAddress = sbus1.msAddress;
            mbus.msData = sbus1.msData;
            mbus.msWrite = sbus1.msWrite;
            mbus.msValid = sbus1.msValid;
            sbus1.msTaken = mbus.msTaken;
            sbus0.msTaken = 0;
        end
    end

    always_ff @ (posedge clk) begin
        if (mbus.msValid && mbus.msTaken) begin
            lru <= select;
        end
    end
endmodule

module BinaryArbiter_SM #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24
    )(
    MemoryBus.Slave sbus0,
    MemoryBus.Slave sbus1,
    MemoryBus.Master mbus
    );
    always_comb begin
        sbus0.smID = mbus.smID;
        sbus0.smData = mbus.smData;
        sbus0.smValid = mbus.smValid;

        sbus1.smID = mbus.smID;
        sbus1.smData = mbus.smData;
        sbus1.smValid = mbus.smValid;

        mbus.smTaken = sbus0.smTaken || sbus1.smTaken;
    end
endmodule
```

```
/* verilator lint_on DECLFILENAME */
```

6.3 BusToAxi.sv

```
/*
 * Used to interface between out custom bus and a standard
 * 32 bit axi4 bus.
*/
module BusToAxi(
    MemoryBus.Slave bus,

    // axi32 master interface
    input logic arready,
    input logic awready,
    input logic bvalid,
    input logic rlast,
    input logic rvalid,
    input logic wready,
    input logic [1:0] bresp,
    input logic [1:0] rresp,
    input logic [5:0] bid,
    input logic [5:0] rid,
    input logic [31:0] rdata,

    output logic arvalid,
    output logic awvalid,
    output logic bready,
    output logic rready,
    output logic wlast,
    output logic wvalid,
    output logic [1:0] arburst,
    output logic [1:0] arlock,
    output logic [2:0] arsize,
    output logic [1:0] awburst,
    output logic [1:0] awlock,
    output logic [2:0] awsize,
    output logic [2:0] arprot,
    output logic [2:0] awprot,
    output logic [31:0] araddr,
    output logic [31:0] awaddr,
    output logic [3:0] arcache,
    output logic [3:0] arlen,
    output logic [3:0] arqos,
    output logic [3:0] awcache,
    output logic [3:0] awlen,
    output logic [3:0] awqos,
```

```
output logic [5:0] arid,
output logic [5:0] awid,
output logic [5:0] wid,
output logic [31:0] wdata,
output logic [3:0] wstrb
);

// convert from axi to bus
assign bus.smID = 8'(rid);
assign bus.smData = rdata[23:0];
assign bus.smValid = rvalid;
logic rx_read_ready = bus.smTaken;

// convert from bus to axi
logic write = bus.msWrite;
logic valid = bus.msValid;
logic ready = write ? awready && wready : arready;
assign bus.msTaken = valid && ready;

logic [5:0] id = bus.msID[5:0];
logic [31:0] data = 32'(bus.msData);
logic [31:0] address = {bus.msAddress[29:0], 2'b0};

// static settings
logic rx_write_ready = 1;
logic protection = 0;
logic cache = 0;

// populate all of the axi outputs
always_comb begin
    arvalid = !write && valid;
    awvalid = write && valid;
    bready = rx_write_ready;
    rready = rx_read_ready;
    wlast = write && valid; // same as wvalid
    wvalid = write && valid;
    arburst = 2'b01;
    arlock = 0;
    arsize = 'b010;
    awburst = 2'b01;
    awlock = 0;
    awsize = 'b010;
    arprot = protection;
    awprot = protection;
    araddr = address;
    awaddr = address;
```

```
        arcache = cache;
        arlen = 0;
        arqos = 0;
        awcache = cache;
        awlen = 0; //burst length of 0
        awqos = 0;
        arid = id;
        awid = id;
        wid = id;
        wdata = data;
        wstrb = 4'b1111;
    end

endmodule: BusToAxi
```

6.4 BusToAxiLite.sv

```
module BusToAxiLite(
    input logic clock,
    input logic reset,

    MemoryBus.Slave bus,

    // axi4lite master interface
    input logic arready,
    input logic awready,
    input logic bvalid,
    input logic rvalid,
    input logic wready,
    input logic [1:0] bresp,
    input logic [1:0] rresp,
    input logic [31:0] rdata,

    output logic arvalid,
    output logic awvalid,
    output logic bready,
    output logic rready,
    output logic wvalid,

    output logic [2:0] arprot,
    output logic [2:0] awprot,
    output logic [31:0] araddr,
    output logic [31:0] awaddr,


    output logic [31:0] wdata,
```

```
   output logic [3:0] wstrb
   );
   logic [5:0] id;
   // convert from axi to bus
   assign bus.smID = 8'(id);
   assign bus.smData = rdata[23:0];
   assign bus.smValid = rvalid;
   logic rx_read_ready = bus.smTaken;

   // convert from bus to axi
   logic write = bus.msWrite;
   logic valid = bus.msValid;
   logic ready = write ? awready && wready : arready;
   assign bus.msTaken = valid && ready;

   logic [31:0] data = 32'(bus.msData);
   logic [31:0] address = {bus.msAddress[29:0], 2'b0};

   // static settings
   logic rx_write_ready = 1;
   logic protection = 0;
   logic cache = 0;

   // populate all of the axi outputs
always_comb begin
       arvalid = !write && valid;
       awvalid = write && valid;
       bready = rx_write_ready;
       rready = rx_read_ready;
       wvalid = write && valid;



       arprot = protection;
       awprot = protection;
       araddr = address;
       awaddr = address;


       wdata = data;
       wstrb = 4'b1111;
   end

always_ff @ (posedge clock) begin
   if (reset) begin
       id <= 0;
```

```
    end else begin
        if (bus.msValid && !bus.msWrite && bus.msTaken) begin
            id <= bus.msID[5:0];
        end
    end
 end
endmodule
```

6.5 Crossbar.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/20/2019 08:56:18 PM
// Design Name:
// Module Name: Crossbar
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Crossbar #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24
    )(
    input clk,
    input rst,
    MemoryBus.Slave ray_m,
    MemoryBus.Slave ps,
    MemoryBus.Master ray_s,
    MemoryBus.Master bram,
    MemoryBus.Master dram
    );
    Crossbar_MS ms(.clk(clk), .rst(rst), .ray_m(ray_m), .ps(ps), .ray_s(ray_s), .bram(bram),
```

```
    Crossbar_SM sm(.clk(clk), .rst(rst), .ray_m(ray_m), .ps(ps), .ray_s(ray_s), .bram(bram),
endmodule

module Crossbar_MS #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24
    )(
    input clk,
    input rst,
    MemoryBus.Slave ray_m,
    MemoryBus.Slave ps,
    MemoryBus.Master ray_s,
    MemoryBus.Master bram,
    MemoryBus.Master dram
    );

    localparam RAY_S_BASE = 32'h0;
    localparam BRAM_BASE = 32'h100;
    localparam DRAM_BASE = 32'h800;

    logic ray_m_ray_s;
    assign ray_m_ray_s = ray_m.msAddress < BRAM_BASE;
    logic ps_ray_s;
    assign ps_ray_s = ps.msAddress < BRAM_BASE;

    logic ray_m_bram;
    assign ray_m_bram = ray_m.msAddress < DRAM_BASE && ray_m.msAddress >= BRAM_BASE;
    logic ps_bram;
    assign ps_bram = ps.msAddress < DRAM_BASE && ps.msAddress >= BRAM_BASE;

    logic ray_m_dram;
    assign ray_m_dram = ray_m.msAddress >= DRAM_BASE;
    logic ps_dram;
    assign ps_dram = ps.msAddress >= DRAM_BASE;


    logic ray_s_lru;
    logic bram_lru;
    logic dram_lru;
    logic[1:0] ray_s_state;
    logic[1:0] bram_state;
    logic[1:0] dram_state;
    localparam INACTIVE = 2'b00;
    localparam PASS_RAY_M = 2'b10;
    localparam PASS_PS = 2'b11;
```

```
localparam LRU_RAY_M = 1'b0;
localparam LRU_PS = 1'b1;

always_comb begin
    //ray_s
    if (ray_m.msValid && ray_m_ray_s && !(ps.msValid && ps_ray_s)) begin
    //pass from ray_m to ray_s
        ray_s.msID = ray_m.msID;
        ray_s.msAddress = ray_m.msAddress - RAY_S_BASE;
        ray_s.msData = ray_m.msData;
        ray_s.msWrite = ray_m.msWrite;
        ray_s.msValid = ray_m.msValid;
        ray_m.msTaken = ray_s.msTaken;
        if (ps_ray_s) begin
            ps.msTaken = 0;
        end
        ray_s_state = PASS_RAY_M;
        //TODO: what to do about ps taken?
    end else if (!(ray_m.msValid && ray_m_ray_s) && ps.msValid && ps_ray_s) begin
    //pass from ps to ray_s
        ray_s.msID = ps.msID;
        ray_s.msAddress = ps.msAddress - RAY_S_BASE;
        ray_s.msData = ps.msData;
        ray_s.msWrite = ps.msWrite;
        ray_s.msValid = ps.msValid;
        ps.msTaken = ray_s.msTaken;
        if (ray_m_ray_s) begin
            ray_m.msTaken = 0;
        end
        ray_s_state = PASS_PS;
    end else if (ray_m.msValid && ray_m_ray_s && ps.msValid && ps_ray_s) begin
        if (ray_s_lru == LRU_RAY_M) begin
        //pass from ray_m to ray_s
            ray_s.msID = ray_m.msID;
            ray_s.msAddress = ray_m.msAddress - RAY_S_BASE;
            ray_s.msData = ray_m.msData;
            ray_s.msWrite = ray_m.msWrite;
            ray_s.msValid = ray_m.msValid;
            ray_m.msTaken = ray_s.msTaken;
            ps.msTaken = 0;
            ray_s_state = PASS_RAY_M;
        end else begin
            ray_s.msID = ps.msID;
            ray_s.msAddress = ps.msAddress - RAY_S_BASE;
            ray_s.msData = ps.msData;
            ray_s.msWrite = ps.msWrite;
```

```
        ray_s.msValid = ps.msValid;
        ps.msTaken = ray_s.msTaken;
        ray_m.msTaken = 0;
        ray_s_state = PASS_PS;
    end
end else begin
    ray_s.msValid = 0;
    ray_s_state = INACTIVE;
    if (ps_ray_s) begin
        ps.msTaken = 0;
    end
    if (ray_m_ray_s) begin
        ray_m.msTaken = 0;
    end
end

//bram
if (ray_m.msValid && ray_m_bram && !(ps.msValid && ps_bram)) begin
//pass from ray_m to bram
    bram.msID = ray_m.msID;
    bram.msAddress = ray_m.msAddress - BRAM_BASE;
    bram.msData = ray_m.msData;
    bram.msWrite = ray_m.msWrite;
    bram.msValid = ray_m.msValid;
    ray_m.msTaken = bram.msTaken;
    if (ps_bram) begin
        ps.msTaken = 0;
    end
    bram_state = PASS_RAY_M;
end else if (!(ray_m.msValid && ray_m_bram) && ps.msValid && ps_bram) begin
//pass from ps to bram
    bram.msID = ps.msID;
    bram.msAddress = ps.msAddress - BRAM_BASE;
    bram.msData = ps.msData;
    bram.msWrite = ps.msWrite;
    bram.msValid = ps.msValid;
    ps.msTaken = bram.msTaken;
    if (ray_m_bram) begin
        ray_m.msTaken = 0;
    end
    bram_state = PASS_PS;
end else if (ray_m.msValid && ray_m_bram && ps.msValid && ps_bram) begin
    if (bram_lru == LRU_RAY_M) begin
    //pass from ray_m to bram
        bram.msID = ray_m.msID;
        bram.msAddress = ray_m.msAddress - BRAM_BASE;
```

```
            bram.msData = ray_m.msData;
            bram.msWrite = ray_m.msWrite;
            bram.msValid = ray_m.msValid;
            ray_m.msTaken = bram.msTaken;
            ps.msTaken = 0;
            bram_state = PASS_RAY_M;
        end else begin
            bram.msID = ps.msID;
            bram.msAddress = ps.msAddress - BRAM_BASE;
            bram.msData = ps.msData;
            bram.msWrite = ps.msWrite;
            bram.msValid = ps.msValid;
            ps.msTaken = bram.msTaken;
            ray_m.msTaken = 0;
            bram_state = PASS_PS;
        end
    end else begin
        bram.msValid = 0;
        bram_state = INACTIVE;
        if (ps_bram) begin
            ps.msTaken = 0;
        end
        if (ray_m_bram) begin
            ray_m.msTaken = 0;
        end
    end

    //dram
    if (ray_m.msValid && ray_m_dram && !(ps.msValid && ps_dram)) begin
    //pass from ray_m to dram
        dram.msID = ray_m.msID;
        dram.msAddress = ray_m.msAddress - DRAM_BASE;
        dram.msData = ray_m.msData;
        dram.msWrite = ray_m.msWrite;
        dram.msValid = ray_m.msValid;
        ray_m.msTaken = dram.msTaken;
        if (ps_dram) begin
            ps.msTaken = 0;
        end
        dram_state = PASS_RAY_M;
    end else if (!(ray_m.msValid && ray_m_dram) && ps.msValid && ps_dram) begin
    //pass from ps to dram
        dram.msID = ps.msID;
        dram.msAddress = ps.msAddress - DRAM_BASE;
        dram.msData = ps.msData;
        dram.msWrite = ps.msWrite;
```

```
            dram.msValid = ps.msValid;
            ps.msTaken = dram.msTaken;
            if (ray_m_dram) begin
                ray_m.msTaken = 0;
            end
            dram_state = PASS_PS;
        end else if (ray_m.msValid && ray_m_dram && ps.msValid && ps_dram) begin
            if (dram_lru == LRU_RAY_M) begin
            //pass from ray_m to dram
                dram.msID = ray_m.msID;
                dram.msAddress = ray_m.msAddress - DRAM_BASE;
                dram.msData = ray_m.msData;
                dram.msWrite = ray_m.msWrite;
                dram.msValid = ray_m.msValid;
                ray_m.msTaken = dram.msTaken;
                ps.msTaken = 0;
                dram_state = PASS_RAY_M;
            end else begin
                dram.msID = ps.msID;
                dram.msAddress = ps.msAddress - DRAM_BASE;
                dram.msData = ps.msData;
                dram.msWrite = ps.msWrite;
                dram.msValid = ps.msValid;
                ps.msTaken = dram.msTaken;
                ray_m.msTaken = 0;
                dram_state = PASS_PS;
            end
        end else begin
            dram.msValid = 0;
            dram_state = INACTIVE;
            if (ps_dram) begin
                ps.msTaken = 0;
            end
            if (ray_m_dram) begin
                ray_m.msTaken = 0;
            end
        end
    end
    always_ff @ (posedge clk) begin
        if (rst) begin
            ray_s_lru <= LRU_PS;
            bram_lru <= LRU_PS;
            dram_lru <= LRU_PS;
        end else begin
            if (ray_s_state == PASS_RAY_M && ray_m.msTaken) begin
                ray_s_lru <= LRU_PS;
```

```
            end else if (ray_s_state == PASS_PS && ps.msTaken) begin
                ray_s_lru <= LRU_RAY_M;
            end
            if (bram_state == PASS_RAY_M && ray_m.msTaken) begin
                bram_lru <= LRU_PS;
            end else if (bram_state == PASS_PS && ps.msTaken) begin
                bram_lru <= LRU_RAY_M;
            end
            if (dram_state == PASS_RAY_M && ray_m.msTaken) begin
                dram_lru <= LRU_PS;
            end else if (dram_state == PASS_PS && ps.msTaken) begin
                dram_lru <= LRU_RAY_M;
            end
        end
    end
endmodule

module Crossbar_SM #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=24
    )(
    input clk,
    input rst,
    MemoryBus.Slave ray_m,
    MemoryBus.Slave ps,
    MemoryBus.Master ray_s,
    MemoryBus.Master bram,
    MemoryBus.Master dram
    );

    logic [1:0] state;
    logic [1:0] mru;

    localparam INACTIVE = 2'b00;
    localparam PASS_RAY_S = 2'b01;
    localparam PASS_BRAM = 2'b10;
    localparam PASS_DRAM = 2'b11;

    localparam MRU_RAY_S = 2'b00;
    localparam MRU_BRAM = 2'b01;
    localparam MRU_DRAM = 2'b10;

    always_comb begin
        if (mru == MRU_RAY_S) begin
            // check bram then dram then ray s
```

```
    if (bram.smValid) begin
        ray_m.smID = bram.smID;
        ray_m.smData = bram.smData;
        ray_m.smValid = bram.smValid;

        ps.smID = bram.smID;
        ps.smData = bram.smData;
        ps.smValid = bram.smValid;

        bram.smTaken = ray_m.smTaken || ps.smTaken;
        dram.smTaken = 0;
        ray_s.smTaken = 0;
        state = PASS_BRAM;
    end else if (dram.smValid) begin
        ray_m.smID = dram.smID;
        ray_m.smData = dram.smData;
        ray_m.smValid = dram.smValid;

        ps.smID = dram.smID;
        ps.smData = dram.smData;
        ps.smValid = dram.smValid;

        dram.smTaken = ray_m.smTaken || ps.smTaken;
        bram.smTaken = 0;
        ray_s.smTaken = 0;
        state = PASS_DRAM;
    end else if (ray_s.smValid) begin
        ray_m.smID = ray_s.smID;
        ray_m.smData = ray_s.smData;
        ray_m.smValid = ray_s.smValid;

        ps.smID = ray_s.smID;
        ps.smData = ray_s.smData;
        ps.smValid = ray_s.smValid;

        ray_s.smTaken = ray_m.smTaken || ps.smTaken;
        dram.smTaken = 0;
        bram.smTaken = 0;
        state = PASS_RAY_S;
    end else begin
        ray_m.smValid = 0;
        ps.smValid = 0;
        state = INACTIVE;
    end
end else if (mru == MRU_BRAM) begin
    // check dram then ray s then bram
```

```
    if (dram.smValid) begin
        ray_m.smID = dram.smID;
        ray_m.smData = dram.smData;
        ray_m.smValid = dram.smValid;

        ps.smID = dram.smID;
        ps.smData = dram.smData;
        ps.smValid = dram.smValid;

        dram.smTaken = ray_m.smTaken || ps.smTaken;
        bram.smTaken = 0;
        ray_s.smTaken = 0;
        state = PASS_DRAM;
    end else if (ray_s.smValid) begin
        ray_m.smID = ray_s.smID;
        ray_m.smData = ray_s.smData;
        ray_m.smValid = ray_s.smValid;

        ps.smID = ray_s.smID;
        ps.smData = ray_s.smData;
        ps.smValid = ray_s.smValid;

        ray_s.smTaken = ray_m.smTaken || ps.smTaken;
        dram.smTaken = 0;
        bram.smTaken = 0;
        state = PASS_RAY_S;
    end else if (bram.smValid) begin
        ray_m.smID = bram.smID;
        ray_m.smData = bram.smData;
        ray_m.smValid = bram.smValid;

        ps.smID = bram.smID;
        ps.smData = bram.smData;
        ps.smValid = bram.smValid;

        bram.smTaken = ray_m.smTaken || ps.smTaken;
        dram.smTaken = 0;
        ray_s.smTaken = 0;
        state = PASS_BRAM;
    end else begin
        ray_m.smValid = 0;
        ps.smValid = 0;
        state = INACTIVE;
    end
end else if (mru == MRU_DRAM) begin
    // check ray s then bram then dram
```

```
        if (ray_s.smValid) begin
            ray_m.smID = ray_s.smID;
            ray_m.smData = ray_s.smData;
            ray_m.smValid = ray_s.smValid;

            ps.smID = ray_s.smID;
            ps.smData = ray_s.smData;
            ps.smValid = ray_s.smValid;

            ray_s.smTaken = ray_m.smTaken || ps.smTaken;
            dram.smTaken = 0;
            bram.smTaken = 0;
            state = PASS_RAY_S;
        end else if (bram.smValid) begin
            ray_m.smID = bram.smID;
            ray_m.smData = bram.smData;
            ray_m.smValid = bram.smValid;

            ps.smID = bram.smID;
            ps.smData = bram.smData;
            ps.smValid = bram.smValid;

            bram.smTaken = ray_m.smTaken || ps.smTaken;
            dram.smTaken = 0;
            ray_s.smTaken = 0;
            state = PASS_BRAM;
        end else if (dram.smValid) begin
            ray_m.smID = dram.smID;
            ray_m.smData = dram.smData;
            ray_m.smValid = dram.smValid;

            ps.smID = dram.smID;
            ps.smData = dram.smData;
            ps.smValid = dram.smValid;

            dram.smTaken = ray_m.smTaken || ps.smTaken;
            bram.smTaken = 0;
            ray_s.smTaken = 0;
            state = PASS_DRAM;
        end else begin
            ray_m.smValid = 0;
            ps.smValid = 0;
            state = INACTIVE;
        end
    end
end
```

```
    always_ff @ (posedge clk) begin
        if (rst) begin
            state <= INACTIVE;
            mru <= MRU_RAY_S;
        end
        if (state == PASS_RAY_S && ray_s.smTaken) begin
            mru <= MRU_RAY_S;
        end else if (state == PASS_BRAM && bram.smTaken) begin
            mru <= MRU_BRAM;
        end else if (state == PASS_DRAM && dram.smTaken) begin
            mru <= MRU_DRAM;
        end
    end

endmodule
```

6.6 MemoryBus.sv

```
/*
 * Packet like memory interface used within the ray
 * tracing system to allow for fast small transfers of data.
 */
interface MemoryBus #(
    parameter MASTER_ID_WIDTH=8,
    parameter ADDRESS_WIDTH=32,
    parameter DATA_WIDTH=16);

    /*
     * ms - master to slave
     *
     * msID - id of the master used by the slave to
     *      send back a read response
     *
     * msReady & msValid - form a ready valid pair to
     *      handshake the master to slave connection
     */
    logic [MASTER_ID_WIDTH-1:0] msID;
    logic [ADDRESS_WIDTH-1:0]   msAddress;
    logic [DATA_WIDTH-1:0]      msData;
    logic                       msWrite;
    logic                       msTaken;
    logic                       msValid;

    /*
     * sm - slave to master
```

```
     */
    logic [MASTER_ID_WIDTH-1:0] smID;
    logic [DATA_WIDTH-1:0]      smData;
    logic                       smTaken;
    logic                       smValid;

    modport Master(
        output  msID,
        output  msAddress,
        output  msData,
        output  msWrite,
        input   msTaken,
        output  msValid,

        input   smID,
        input   smData,
        output  smTaken,
        input   smValid
    );

    modport Slave(
        input   msID,
        input   msAddress,
        input   msData,
        input   msWrite,
        output  msTaken,
        input   msValid,

        output  smID,
        output  smData,
        input   smTaken,
        output  smValid
    );
endinterface: MemoryBus
```

6.7 MemoryMaster.sv

```
module MemoryMaster(
    input clock,
    input reset,

    // first 24 bits are field, last 8 are control
    input logic [31:0] in,
    output logic [31:0] out,

    MemoryBus.Master bus
```

```
);

logic [23:0] field;
enum logic [7:0] {
    NONE,
    ADDRESS_LOWER,
    ADDRESS_UPPER,
    DATA,
    ID,
    WRITE,
    SEND,
    GET_PENDING,
    GET_DATA,
    GET_ID,
    GET_VALID,
    CLEAR
} command;
assign {command, field} = in;

logic [23:0] lowerAddress;
logic [7:0] upperAddress;
assign bus.msAddress = {upperAddress, lowerAddress};
logic sent;

logic [23:0] rx_data;
logic [7:0] rx_id;
logic rx_valid;

always_comb begin
    case (command) inside
        GET_PENDING: out = 32'(bus.msValid);
        GET_DATA: out = 32'(rx_data);
        GET_ID: out = 32'(rx_id);
        GET_VALID: out = 32'(rx_valid);
        default: out = 32'hxxxxxxxx;
    endcase

    bus.smTaken = !rx_valid && (bus.smID == bus.msID);
end

always_ff @(posedge clock) begin
    if (reset) begin
        bus.msValid <= 0;
        rx_valid <= 0;
        sent <= 0;
    end else begin
```

```
            if (bus.msValid && bus.msTaken) begin
                bus.msValid <= 0;
                sent <= 1;
            end else if (command == SEND && !sent) begin
                bus.msValid <= 1;
            end

            if (command != SEND) begin
                sent <= 0;
            end

            if (bus.smValid && bus.smTaken) begin
                rx_data <= bus.smData;
                rx_id <= bus.smID;
                rx_valid <= 1;
            end else if (command == CLEAR) begin
                rx_valid <= 0;
            end

            case (command)
                ADDRESS_LOWER: lowerAddress <= field;
                ADDRESS_UPPER: upperAddress <= field[7:0];
                DATA: bus.msData <= field;
                ID: bus.msID <= field[7:0];
                WRITE: bus.msWrite <= field[0];
                default: ;
            endcase
        end
    end
endmodule: MemoryMaster
```

6.8 RayConfig.sv

```
module RayConfig #(
    parameter POSITION_WIDTH=16,

    parameter DATA_WIDTH=24,
    parameter ADDRESS_WIDTH=32,

    parameter ADDRESS=0,
    parameter BASE_WIDTH=5
    )(
    input logic clock,
    input logic reset,

    output logic [ADDRESS_WIDTH-1:0] materialAddress,
```

```
output logic [ADDRESS_WIDTH-1:0] treeAddress,
output logic [ADDRESS_WIDTH-1:0] frameAddress,

output logic [POSITION_WIDTH-1:0] cameraQ [2:0],
output logic signed [POSITION_WIDTH-1:0] cameraV [2:0],
output logic signed [POSITION_WIDTH-1:0] cameraX [2:0],
output logic signed [POSITION_WIDTH-1:0] cameraY [2:0],

output logic [11:0] width,
output logic [11:0] height,

output logic start,
input logic ready,
input logic busy,

output logic flush,
output logic resetRT,
output logic normalize,

output logic interrupt,

MemoryBus.Slave bus
);

logic lastBusy;
logic recieved;
logic sent;
logic queuedTransaction;

always_comb begin
    bus.msTaken = {bus.msAddress[ADDRESS_WIDTH-1:BASE_WIDTH], BASE_WIDTH'(0)} == ADDRESS
        && bus.msValid
        && !queuedTransaction;
    bus.smValid = queuedTransaction;

    recieved = bus.msValid && bus.msTaken;
    sent = bus.smValid && bus.smTaken;

    if (recieved && bus.msWrite && bus.msAddress[BASE_WIDTH-1:0] == 0) begin
        start = bus.msData[0];
        flush = bus.msData[3];
        resetRT = bus.msData[4];
    end else begin
        start = 0;
        flush = 0;
        resetRT = reset;
```

```
    end

    interrupt = !busy && lastBusy;
end

always_ff @(posedge clock) begin
    lastBusy <= busy;

    if (reset) begin
        queuedTransaction <= 0;
        normalize <= 0;
    end else if (queuedTransaction) begin
        if (sent) begin
            queuedTransaction <= 0;
        end
    end else if (recieved && bus.msWrite) begin
        case (bus.msAddress[BASE_WIDTH-1:0])
            'h0: normalize <= bus.msData[5];
            'h1: materialAddress <= {bus.msData, 8'b0};
            'h2: treeAddress <= {bus.msData, 8'b0};
            'h3: frameAddress <= {bus.msData, 8'b0};
            'h4: cameraQ[0] <= POSITION_WIDTH'(bus.msData);
            'h5: cameraQ[1] <= POSITION_WIDTH'(bus.msData);
            'h6: cameraQ[2] <= POSITION_WIDTH'(bus.msData);
            'h7: cameraV[0] <= POSITION_WIDTH'(bus.msData);
            'h8: cameraV[1] <= POSITION_WIDTH'(bus.msData);
            'h9: cameraV[2] <= POSITION_WIDTH'(bus.msData);
            'ha: cameraX[0] <= POSITION_WIDTH'(bus.msData);
            'hb: cameraX[1] <= POSITION_WIDTH'(bus.msData);
            'hc: cameraX[2] <= POSITION_WIDTH'(bus.msData);
            'hd: cameraY[0] <= POSITION_WIDTH'(bus.msData);
            'he: cameraY[1] <= POSITION_WIDTH'(bus.msData);
            'hf: cameraY[2] <= POSITION_WIDTH'(bus.msData);
            'h10: width <= 12'(bus.msData);
            'h11: height <= 12'(bus.msData);
        endcase
    end else if (recieved && !bus.msWrite) begin
        case (bus.msAddress[BASE_WIDTH-1:0])
            'h0: bus.smData <= DATA_WIDTH'({normalize, 2'b0, busy, ready, 1'b0});
            'h1: bus.smData <= materialAddress[31:8];
            'h2: bus.smData <= treeAddress[31:8];
            'h3: bus.smData <= frameAddress[31:8];
            'h4: bus.smData <= DATA_WIDTH'(cameraQ[0]);
            'h5: bus.smData <= DATA_WIDTH'(cameraQ[1]);
            'h6: bus.smData <= DATA_WIDTH'(cameraQ[2]);
            'h7: bus.smData <= DATA_WIDTH'(cameraV[0]);
```

```
                'h8: bus.smData <= DATA_WIDTH'(cameraV[1]);
                'h9: bus.smData <= DATA_WIDTH'(cameraV[2]);
                'ha: bus.smData <= DATA_WIDTH'(cameraX[0]);
                'hb: bus.smData <= DATA_WIDTH'(cameraX[1]);
                'hc: bus.smData <= DATA_WIDTH'(cameraX[2]);
                'hd: bus.smData <= DATA_WIDTH'(cameraY[0]);
                'he: bus.smData <= DATA_WIDTH'(cameraY[1]);
                'hf: bus.smData <= DATA_WIDTH'(cameraY[2]);
                'h10: bus.smData <= DATA_WIDTH'(width);
                'h11: bus.smData <= DATA_WIDTH'(height);
            endcase
            queuedTransaction <= 1;
            bus.smID <= bus.msID;
        end
    end

endmodule: RayConfig
```

6.9 RayGenerator.sv

```
/*
 * Module for generating camera rays and driving the
 * ray units.
 */
module RayGenerator #(
    parameter POSITION_WIDTH=16,
    parameter ADDRESS_WIDTH=32
    )(
    input logic clock,
    input logic reset,

    input logic start,
    output logic busy,
    output logic ready,
    input logic normalize,

    input logic signed [POSITION_WIDTH-1:0] cameraV [2:0],
    input logic signed [POSITION_WIDTH-1:0] cameraX [2:0],
    input logic signed [POSITION_WIDTH-1:0] cameraY [2:0],

    input logic [11:0] width,
    input logic [11:0] height,

    // address of the start of the frame
    input logic [ADDRESS_WIDTH-1:0] frameAddress,
```

```
// output to ray units
output logic [POSITION_WIDTH-1:0] rayV [2:0],
output logic [ADDRESS_WIDTH-1:0] rayAddress,
output logic rayStart,
input logic rayReady,
input logic rayBusy
);

enum logic {
    IDLE,
    GENERATING
} state;

logic [11:0] nextX;
logic [11:0] nextY;

logic signed [POSITION_WIDTH-1:0] genV [2:0];
logic signed [POSITION_WIDTH-1:0] regenV [2:0];

logic [11:0] x [9:0];
logic [11:0] y [9:0];
logic valid [10:0];
logic anyBusy [9:0];

logic advance;

// leading zero pairs
logic [3:0] lzp [7:0];
logic [3:0] lza [7:0];
logic [3:0] toShift;

logic h [7:0];
logic l [7:0];

logic signed [31:0] norm2;
logic signed [31:0] y0, y1, y2;
logic signed [31:0] y0_norm2, y1_norm2;
logic signed [31:0] ny0, ny1;
logic signed [31:0] ny0_y0, ny1_y1;
logic signed [31:0] ny0_norm2;
logic signed [31:0] nyy0, nyy1;
logic signed [31:0] nyy0_y0, nyy1_y1;
logic signed [31:0] nyy0_norm2;

always_comb begin
    nextX = x[0] + 1;
```

```
    nextY = y[0] + 1;

    for (int i = 0; i < 8; i++) begin
        h[i] = norm2[15 - 2*i];
        l[i] = norm2[14 - 2*i];
    end

    // zero counting
    for (int i = 0; i < 8; i++) begin
        lzp[i] = {2'b0, !h[i] & !l[i], !h[i] & l[i]};
    end

    lza[7] = lzp[7];
    for (int i = 6; i >= 0; i--) begin
        lza[i] = lzp[i] == 2 ? lzp[i] + lza[i+1] : lzp[i];
    end
    toShift = (lza[0] + 1)>>1;

    advance = rayReady;
    rayStart = valid[10];

    anyBusy[9] = rayStart;
    for (int i = 0; i < 9; i++) begin
        anyBusy[i] = valid[i] || anyBusy[i + 1];
    end
    busy = anyBusy[0] || rayBusy || state != IDLE ;
    ready = state == IDLE;
end

// normalization pipeline
always_ff @(posedge clock) begin
    if (reset) begin
        state <= IDLE;
    end else if (state == IDLE) begin
        if (start) begin
            state <= GENERATING;
            x[0] <= 0;
            y[0] <= 0;
            valid[0] <= 1;
        end
    end else if (state == GENERATING) begin
        if (advance) begin
            if (nextX < width) begin
                x[0] <= nextX;
            end else if (nextY < height) begin
                y[0] <= nextY;
```

```
            x[0] <= 0;
        end else begin
            state <= IDLE;
            valid[0] <= 0;
        end
    end
end

if (advance || reset) begin
    for (int i = 0; i < 9; i++) begin
        x[i + 1] <= x[i];
        y[i + 1] <= y[i];
    end
    for (int i = 0; i < 10; i++) begin
        valid[i + 1] <= reset ? 0 : valid[i];
    end

    // (0) 0. find vector (not carried forth)
    for (int i = 0; i < 3; i++) begin
        genV[i] <= 16'(cameraX[i] * signed'(x[0] -  width/2))
                 + 16'(cameraY[i] * signed'(height/2 - y[0]))
                 + cameraV[i];
    end

    // (1) 1. calculate norm
    norm2 <= (genV[0]*genV[0]
            + genV[1]*genV[1]
            + genV[2]*genV[2]) >> 16;

    // (2) 2. find guess
    y0 <= 1 << 5'(toShift);

    y0_norm2 <= norm2;

    // (3) 3.1 mult step 1
    ny0 <= y0_norm2 * y0;

    ny0_y0 <= y0;
    ny0_norm2 <= y0_norm2;

    // (4) 3.2 mult step 2 and subtraction
    nyy0 <= (3<<16) - (ny0 * ny0_y0);

    nyy0_y0 <= ny0_y0;
    nyy0_norm2 <= ny0_norm2;
```

```
        // (5) 3.3 mult step 3 and drop lower
        y1 <= (nyy0_y0 * nyy0 >> 17);

        y1_norm2 <= nyy0_norm2;

        // (6) 4.1
        ny1 <= y1_norm2 * y1;

        ny1_y1 <= y1;

        // (7) 4.2
        nyy1 <= (3<<16) - (ny1 * ny1_y1);

        nyy1_y1 <= ny1_y1;

        // (8) 4.3
        y2 <= (nyy1_y1 * nyy1 >> 17);

        for (int i = 0; i < 3; i++) begin
            regenV[i] <= 16'(cameraX[i] * signed'(x[8] -  width/2))
                       + 16'(cameraY[i] * signed'(height/2 - y[8]))
                       + cameraV[i];
        end

        // (9) regenerate ray
        for (int i = 0; i < 3; i++) begin
            rayV[i] <= normalize ? 16'(y2 * regenV[i] * 15 / 16) : regenV[i];
        end
        rayAddress <= ADDRESS_WIDTH'(x[9]) + y[9]*width + frameAddress;
    end
end

if (0) begin
    always_ff @(posedge clock) begin
        $display("state: %d, advance: %b", state, advance);
        for (int i = 0; i < 10; i++) begin
            $display("%d = x: %d, y: %d, valid: %d", i, x[i], y[i], valid[i]);
        end
        $display("(2) y0: %d, norm2: %d, lza: %d", y0, y0_norm2, lza[0]);
        $display("(2.lz) a: {%d, %d, %d, %d, %d, %d, %d, %d}, p: {%d, %d, %d, %d, %d, %d
            lza[0], lza[1], lza[2], lza[3], lza[4], lza[5], lza[6], lza[7],
            lzp[0], lzp[1], lzp[2], lzp[3], lzp[4], lzp[5], lzp[6], lzp[7]);
        $display("(2.hl) h: {%d, %d, %d, %d, %d, %d, %d, %d}, l: {%d, %d, %d, %d, %d, %d
            h[0], h[1], h[2], h[3], h[4], h[5], h[6], h[7],
            l[0], l[1], l[2], l[3], l[4], l[5], l[6], l[7]);
        $display("(3) ny0: %d, y0: %d, norm2: %d", ny0, ny0_y0, ny0_norm2);
```

```
            $display("(4) nyy0: %d, y0: %d, norm2: %d", nyy0, nyy0_y0, nyy0_norm2);
            $display("(5) y1: %d, norm2: %d", y1, y1_norm2);
            $display("(6) ny1: %d, y1: %d", ny1, ny1_y1);
            $display("(7) nyy1: %d, y1: %d", nyy1, nyy1_y1);
            $display("(8) y2: %d, regenV: {%d, %d, %d}",
                y2, regenV[0], regenV[1], regenV[2]);
            $display("(9) valid: %d, rayV: {%d, %d, %d}, address: %d\n",
                valid[10], rayV[0], rayV[1], rayV[2], rayAddress);
        end
    end

endmodule: RayGenerator
```

6.10 RayMemory.sv

```
module RayMemory #(
    parameter POSITION_WIDTH=16,

    parameter DATA_WIDTH=24,
    parameter ADDRESS_WIDTH=32,

    parameter MASTER_ID=0,

    parameter MATERIAL_ADDRESS_WIDTH=8
    )(
    input clock,
    input reset,

    // These addresses are the base addresses
    // used for finding the data structures in
    // memory. They are expected to remain constant
    // while the module is in use.
    input logic [ADDRESS_WIDTH-1:0] materialAddress,
    input logic [ADDRESS_WIDTH-1:0] treeAddress,

    // flush out the octree caching, should only be
    // called when the system is idleing.
    input logic flush,

    // Control signal to start traversing the octree and find
    // the depth and material at the position.
    input logic traverse,
    input logic outOfBounds,
    input logic [POSITION_WIDTH-1:0] position [2:0],
    // Depth in the octree where 0 is the root node, and
    // each addition number is one node further down in the
```

```
// tree. Used to calculate the bounding box of the leaf
// node.
output logic [3:0] depth,
// Material properties at that leaf node. Currently this
// width is limited to a single data width, but it can be
// easily extended in the future.
output logic [DATA_WIDTH-1:0] material,

// Control signal used to write a 'pixel' pixel to pixelAddress
input logic writePixel,
input logic [23:0] pixel,
input logic [ADDRESS_WIDTH-1:0] pixelAddress,

output logic ready,

MemoryBus.Master bus
);

parameter TREE_OCTANT_SELECT = 3;
parameter TREE_NODE_ADDRESS_SIZE = DATA_WIDTH;

parameter MATERIAL_MASK_SIZE = DATA_WIDTH - MATERIAL_ADDRESS_WIDTH;

// IDLE - system is ready for a new command
//
// PIXEL_SEND - used to potentially extend the
//      signal if it is not ready to immediatly be
//      sent, then turn off valid once it is
//
// TRAVERSE_SEND - similar to pixel send
//
// TRAVERSE_RECIEVE - wait on requested data
//
// MARTERIAL_(SEND/RECIEVE) - similar to TRAVERSE
enum logic[2:0] {
    IDLE,
    CACHE_SEARCH,
    PIXEL_SEND,
    TRAVERSE_SEND,
    TRAVERSE_RECIEVE,
    MATERIAL_SEND,
    MATERIAL_RECIEVE
} state;

// used to select the correct octant from an octree node
logic [TREE_OCTANT_SELECT-1:0] octantSelect;
```

```
logic [POSITION_WIDTH-1:0] cacheQ [2:0];
logic [3:0] cacheDepth;
logic [DATA_WIDTH-1:0] cacheNode [POSITION_WIDTH-1:0];

// set whether bus is currently receiving
logic busReceiving;
logic busReceived;
logic busSending;
logic busSent;

logic [3:0] dSelect;

always_comb begin
    ready = state == IDLE;
    /* verilator lint_off WIDTH */
    bus.msID = MASTER_ID;
    /* verilator lint_on WIDTH */

    // selecting the bits at depth, then packing them
    // into a value that will be used to select the octant
    // in each octree node
    dSelect = 4'(POSITION_WIDTH - 1) - depth;
    octantSelect =
        {
            position[2][dSelect],
            position[1][dSelect],
            position[0][dSelect]
        };

    busReceiving =
           state == TRAVERSE_RECIEVE
        || state == MATERIAL_RECIEVE;

    /* verilator lint_off WIDTH */
    bus.smTaken =
           busReceiving
        && (bus.smID == MASTER_ID)
        && bus.smValid;
    busReceived = bus.smTaken && bus.smValid;
    /* verilator lint_on WIDTH */

    busSending =
           state == TRAVERSE_SEND
        || state == MATERIAL_SEND
        || state == PIXEL_SEND;
```

```
    bus.msValid = busSending;
    busSent = bus.msTaken && bus.msValid;
end

always_ff @(posedge clock) begin
    if (reset) begin
        state <= IDLE;
        cacheDepth <= 0;
    end else if (flush) begin
        cacheDepth <= 0;
    end else if (state == IDLE) begin
        if (writePixel) begin
            state <= PIXEL_SEND;

            bus.msData <= DATA_WIDTH'(pixel);
            bus.msAddress <= pixelAddress;
            bus.msWrite <= 1;
        end else if (traverse) begin
            if (outOfBounds) begin
                state <= MATERIAL_SEND;
                bus.msAddress <= materialAddress;
            end else begin
                state <= CACHE_SEARCH;
                depth <= 0;
            end
        end
    end else if (state == CACHE_SEARCH) begin
        if (depth >= cacheDepth
            || cacheQ[0][dSelect] != position[0][dSelect]
            || cacheQ[1][dSelect] != position[1][dSelect]
            || cacheQ[2][dSelect] != position[2][dSelect]) begin
            state <= TRAVERSE_SEND;

            bus.msAddress <= treeAddress + (depth == 0 ?
                ADDRESS_WIDTH'(octantSelect) :
                ADDRESS_WIDTH'({cacheNode[depth-1] , octantSelect}));
            bus.msWrite <= 0;
        end

        depth <= depth + 1;
    end else if (state == PIXEL_SEND) begin
        if (busSent) begin
            state <= IDLE;
        end
    end else if (state == TRAVERSE_SEND) begin
        if (busSent) begin
```

```
        state <= TRAVERSE_RECIEVE;
    end
end else if (state == TRAVERSE_RECIEVE) begin
    if (busReceived) begin
        if (bus.smData[DATA_WIDTH-1:DATA_WIDTH-MATERIAL_MASK_SIZE] == '1) begin
            // this is a material
            // check if this is spacial case 0
            if (bus.smData[MATERIAL_ADDRESS_WIDTH-1:0] == 0) begin
                state <= IDLE;
                material <= 0;
            end else begin
                state <= MATERIAL_SEND;
                bus.msAddress <= materialAddress +
                    ADDRESS_WIDTH'(bus.smData[MATERIAL_ADDRESS_WIDTH-1:0]);
            end

            cacheDepth <= depth - 1;
            cacheQ[0] <= position[0];
            cacheQ[1] <= position[1];
            cacheQ[2] <= position[2];
        end else begin
            // this is a node
            state <= TRAVERSE_SEND;
            bus.msAddress <= treeAddress +
                ADDRESS_WIDTH'({bus.smData, octantSelect});

            // save to cache
            cacheNode[depth-1] <= bus.smData;

            depth <= depth + 1;
        end
    end
end else if (state == MATERIAL_SEND) begin
    if (busSent) begin
        state <= MATERIAL_RECIEVE;
    end
end else if (state == MATERIAL_RECIEVE) begin
    if (busReceived) begin
        state <= IDLE;
        material <= bus.smData;
    end
end

if (0) begin
    $display("state: %d, msValid: %d, msTaken: %d, msAddress: %d, msData: %d, smVali
        state, bus.msValid, bus.msTaken, bus.msAddress, bus.msData, bus.smValid, bus
```

```
        end
        if (0) begin
            $display("octant: %b, depth: %d, position: {%d, %d, %d}, cacheDepth: %d",
                octantSelect, depth, position[0], position[1], position[2], cacheDepth);
        end
    end
endmodule: RayMemory
```

6.11 RayStepper.sv

```
/*
 * This module is a deterministic ray stepper. It will take in a ray
 * (a position and a direction) and an AABB then return the position
 * along the ray where it just exits the bounding box.
 */

module RayStepper #(
    parameter WIDTH=16
    )(
    input clock,
    input reset,

    // resets and starts new operation
    // operation begins after start returns to 0
    input start,
    // q and v are latched when start is high
    // q is the initial position
    // v is the signed direction vector and must have a length
    // in (sqrt(3)/2, 1). This assumption allows faster searching
    input [WIDTH-1:0] q [2:0],
    input [WIDTH-1:0] v [2:0],
    // l and u must be held constant during the operation
    // l is the lower bound on each axis
    // u is the upper bound on each axis
    input [WIDTH-1:0] l [2:0],
    input [WIDTH-1:0] u [2:0],

    // indicates that the ray exited the maximum width
    output logic outOfBounds,
    output logic done,
    // gives the new position
    output logic [WIDTH-1:0] qp [2:0]
    );

    logic [WIDTH-1:0] accumulator [2:0];
    // 1 extra bit for to round up
```

```
logic signed [WIDTH:0] step [2:0];

logic signed [WIDTH-1:0] roundedStep [2:0];

// 2 extra bits for out of bounds detection
// in bounds is when 2 MSB are both zero
logic signed [WIDTH+1:0] proposedPosition [2:0];
logic signed [WIDTH+1:0] lMinusOne [2:0];
logic signed [WIDTH+1:0] uPlusOne [2:0];

logic inAABB [2:0];
logic onAABB [2:0];

always_comb begin
    for (int i = 0; i < 3; i++) begin
        // round if odd and positive
        roundedStep[i] = step[i][WIDTH:1] +
            (!step[i][WIDTH] && step[i][0] ? 1 : 0);
        // sign extend the step value
        proposedPosition[i] = {2'b0, accumulator[i]} +
            {{2{roundedStep[i][WIDTH-1]}}, roundedStep[i]};

        lMinusOne[i] = {2'b0, l[i]} - 1;
        uPlusOne[i]  = {2'b0, u[i]} + 1;

        inAABB[i] = lMinusOne[i] <= proposedPosition[i]
                                    && proposedPosition[i] <= uPlusOne[i];
        onAABB[i] = lMinusOne[i] == proposedPosition[i]
                                    || proposedPosition[i] == uPlusOne[i];

        qp[i] = accumulator[i];
    end
end

always_ff @(posedge clock) begin
    if (reset) begin
        done <= 1;
        outOfBounds <= 0;
    end else if (start) begin
        // load registers
        for (int i = 0; i < 3; i++) begin
            step[i] <= {v[i], 1'b0};
            accumulator[i] <= q[i];
        end

        // start
```

```
            done <= 0;
            outOfBounds <= 0;
        end else if (!done) begin
            if (0) begin
                $display("on: %d, %d, %d", onAABB[0], onAABB[1], onAABB[2]);
                $display("in: %d, %d, %d", inAABB[0], inAABB[1], inAABB[2]);
                $display("lower: %d, %d, %d", l[0], l[1], l[2]);
                $display("upper: %d, %d, %d", u[0], u[1], u[2]);
                $display("working: %d, %d, %d", accumulator[0], accumulator[1], accumulator
                $display("step: %d, %d, %d", step[0]/2, step[1]/2, step[2]/2);
                $display("pos: %d, %d, %d\n", proposedPosition[0], proposedPosition[1], prop
            end

            // commit changes if they fit
            if (inAABB[0] && inAABB[1] && inAABB[2]) begin
                for (int i = 0; i < 3; i++) begin
                    accumulator[i] <= proposedPosition[i][WIDTH-1:0];
                end

                if (onAABB[0] || onAABB[1] || onAABB[2]) begin
                    done <= 1;
                    if (proposedPosition[0][WIDTH+1:WIDTH] != 0 ||
                        proposedPosition[1][WIDTH+1:WIDTH] != 0 ||
                        proposedPosition[2][WIDTH+1:WIDTH] != 0) begin
                        outOfBounds <= 1;
                    end
                end
            end

            for (int i = 0; i < 3; i++) begin
                step[i] <= {step[i][WIDTH], step[i][WIDTH:1]};
            end
        end
    end

endmodule: RayStepper
```

6.12 RayTracer.sv

```
module RayTracer #(
    parameter POSITION_WIDTH=16,

    parameter DATA_WIDTH=24,
    parameter ADDRESS_WIDTH=32,

    parameter CONFIG_ADDRESS=0,
```

```
parameter MASTER_ID_BASE=0
)(
input logic clock,
input logic reset,

output logic interrupt,

MemoryBus.Slave configPort,
MemoryBus.Master memoryPort
);

logic [ADDRESS_WIDTH-1:0] materialAddress;
logic [ADDRESS_WIDTH-1:0] treeAddress;
logic [ADDRESS_WIDTH-1:0] frameAddress;

logic [POSITION_WIDTH-1:0] cameraQ [2:0];
logic signed [POSITION_WIDTH-1:0] cameraV [2:0];
logic signed [POSITION_WIDTH-1:0] cameraX [2:0];
logic signed [POSITION_WIDTH-1:0] cameraY [2:0];

logic [11:0] width;
logic [11:0] height;

logic start;
logic ready;
logic busy;

logic flush;
logic resetRT;
logic normalize;

RayConfig #(
    .POSITION_WIDTH(POSITION_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .ADDRESS_WIDTH(ADDRESS_WIDTH),
    .ADDRESS(CONFIG_ADDRESS)
) cnfg (
    .clock(clock),
    .reset(reset),
    .materialAddress(materialAddress),
    .treeAddress(treeAddress),
    .frameAddress(frameAddress),
    .cameraQ(cameraQ),
    .cameraV(cameraV),
    .cameraX(cameraX),
    .cameraY(cameraY),
```

```
    .width(width),
    .height(height),
    .start(start),
    .ready(ready),
    .busy(busy),
    .flush(flush),
    .resetRT(resetRT),
    .normalize(normalize),
    .interrupt(interrupt),
    .bus(configPort));

logic [POSITION_WIDTH-1:0] rayV [2:0];
logic [ADDRESS_WIDTH-1:0] rayAddress;
logic rayStart;
logic rayReady;
logic rayBusy;

RayGenerator #(
    .POSITION_WIDTH(POSITION_WIDTH),
    .ADDRESS_WIDTH(ADDRESS_WIDTH)
) generator (
    .clock(clock),
    .reset(resetRT),
    .start(start),
    .busy(busy),
    .ready(ready),
    .normalize(normalize),
    .cameraV(cameraV),
    .cameraX(cameraX),
    .cameraY(cameraY),
    .width(width),
    .height(height),
    .frameAddress(frameAddress),
    .rayV(rayV),
    .rayAddress(rayAddress),
    .rayStart(rayStart),
    .rayReady(rayReady),
    .rayBusy(rayBusy));

RayUnit #(
    .POSITION_WIDTH(POSITION_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .ADDRESS_WIDTH(ADDRESS_WIDTH),
    .MASTER_ID(MASTER_ID_BASE)
) unit (
    .clock(clock),
```

```
        .reset(resetRT),
        .flush(flush),
        .start(rayStart),
        .busy(rayBusy),
        .ready(rayReady),
        .rayQ(cameraQ),
        .rayV(rayV),
        .pixelAddress(rayAddress),
        .materialAddress(materialAddress),
        .treeAddress(treeAddress),
        .bus(memoryPort));

endmodule: RayTracer
```

6.13 RayUnit.sv

```
/*
 * Top level container for a ray unit
 */
module RayUnit #(
    parameter POSITION_WIDTH=16,
    parameter DATA_WIDTH=24,
    parameter ADDRESS_WIDTH=32,
    parameter MASTER_ID=0
    )(
    input logic clock,
    input logic reset,

    // flush cached data
    input logic flush,

    // ready/valid pair for controlling ray unit
    // TODO: consider seperate busy and ready signals
    // so that the ray tracer management system can
    // know when the system is finished with a frame.
    input logic start,
    output logic busy,
    output logic ready,

    // incoming ray position and direction
    // the ray position is garunteed to remain stable
    // while busy
    input logic [POSITION_WIDTH-1:0] rayQ [2:0],
    input logic [POSITION_WIDTH-1:0] rayV [2:0],

    // pixel address to write out to, it is saved on the
```

```
// start signal
input logic [ADDRESS_WIDTH-1:0] pixelAddress,

// addresses for finding the material and tree
// these are garunteed to remain stable while the
// ray unit is busy
input logic [ADDRESS_WIDTH-1:0] materialAddress,
input logic [ADDRESS_WIDTH-1:0] treeAddress,

MemoryBus.Master bus
);
enum logic[2:0] {
    IDLE,
    STEP_START,
    STEP_FINISH,
    TRAVERSE_START,
    TRAVERSE_FINISH,
    WRITE
} state;

logic [POSITION_WIDTH-1:0] q [2:0];
logic [POSITION_WIDTH-1:0] qp [2:0];
logic [POSITION_WIDTH-1:0] v [2:0];

logic [POSITION_WIDTH-1:0] l [2:0];
logic [POSITION_WIDTH-1:0] u [2:0];

logic [POSITION_WIDTH-1:0] mask;

logic [3:0] depth;

logic step;
logic stepReady;

logic traverse;
logic traverseReady;

logic write;
logic writeReady;

logic memoryReady;
logic outOfBounds;

logic getBackground;

logic [DATA_WIDTH-1:0] material;
```

```
logic [DATA_WIDTH-1:0] pixel;

logic [ADDRESS_WIDTH-1:0] pixelAddressF;

always_comb begin
    // find l and u
    mask = (1 << (POSITION_WIDTH - depth)) - 1;
    for (int i = 0; i < 3; i++) begin
        l[i] = q[i] & ~mask;
        u[i] = q[i] | mask;
    end

    write = state == WRITE;
    step = state == STEP_START;
    traverse = state == TRAVERSE_START;

    // so that these can be decoupled later
    writeReady = memoryReady;
    traverseReady = memoryReady & !write;

    pixel = material;

    busy = state != IDLE;
    ready = !busy;
end

always_ff @(posedge clock) begin
    if (0) begin
        $display("state: %d, traverse: %d, tReady: %d, x: %d, y: %d, z: %d", state, trav
        $display("mask: %b, x: [%d, %d], y: [%d, %d], z: [%d, %d]", mask, l[0], u[0], l
    end

    if (reset) begin
        state <= IDLE;
    end else if (state == IDLE) begin
        if (start) begin
            state <= TRAVERSE_START;

            pixelAddressF <= pixelAddress;
            for (int i = 0; i < 3; i++) begin
                q[i] <= rayQ[i];
                v[i] <= rayV[i];
            end

            getBackground <= 0;
        end
```

```
    end else if (state == TRAVERSE_START) begin
        if (traverseReady) begin
            state <= TRAVERSE_FINISH;
        end
    end else if (state == TRAVERSE_FINISH) begin
        if (traverseReady) begin
            if (material == 0 && !getBackground) begin
                state <= STEP_START;
            end else begin
                state <= WRITE;
            end
        end
    end else if (state == STEP_START) begin
        if (stepReady) begin
            state <= STEP_FINISH;
        end
    end else if (state == STEP_FINISH) begin
        if (stepReady) begin
            getBackground <= outOfBounds;

            state <= TRAVERSE_START;
            q <= qp;
        end
    end else if (state == WRITE) begin
        if (writeReady) begin
            state <= IDLE;
        end
    end
end

// TODO: seperate ready done interfaces
// TODO: unify signal names between modules
RayStepper#(
    .WIDTH(POSITION_WIDTH)
    ) stepper(
    .clock(clock),
    .reset(reset),
    .start(step),
    .q(q),
    .v(v),
    .l(l),
    .u(u),
    .outOfBounds(outOfBounds),
    .done(stepReady),
    .qp(qp));
```

```
    RayMemory#(
        .POSITION_WIDTH(POSITION_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .ADDRESS_WIDTH(ADDRESS_WIDTH),
        .MASTER_ID(MASTER_ID),
        .MATERIAL_ADDRESS_WIDTH(8)
        ) memory(
        .clock(clock),
        .reset(reset),
        .materialAddress(materialAddress),
        .treeAddress(treeAddress),
        .flush(flush),
        .traverse(traverse),
        .outOfBounds(getBackground),
        .position(q),
        .depth(depth),
        .material(material),
        .writePixel(write),
        .pixel(pixel),
        .pixelAddress(pixelAddressF),
        .ready(memoryReady),
        .bus(bus));

endmodule: RayUnit
```

6.14 System_xbar.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/05/2019 05:54:29 PM
// Design Name:
// Module Name: System_xbar
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
////////////////////////////////////////////////////////////////////////////////


module System_xbar(
    input logic clock,
    input logic reset,

    input logic [31:0] gpio_in,
    output logic [31:0] gpio_out,

    input logic arready,
    input logic awready,
    input logic bvalid,
    input logic rlast,
    input logic rvalid,
    input logic wready,
    input logic [1:0] bresp,
    input logic [1:0] rresp,
    input logic [5:0] bid,
    input logic [5:0] rid,
    input logic [31:0] rdata,

    output logic arvalid,
    output logic awvalid,
    output logic bready,
    output logic rready,
    output logic wlast,
    output logic wvalid,
    output logic [1:0] arburst,
    output logic [1:0] arlock,
    output logic [2:0] arsize,
    output logic [1:0] awburst,
    output logic [1:0] awlock,
    output logic [2:0] awsize,
    output logic [2:0] arprot,
    output logic [2:0] awprot,
    output logic [31:0] araddr,
    output logic [31:0] awaddr,
    output logic [3:0] arcache,
    output logic [3:0] arlen,
    output logic [3:0] arqos,
    output logic [3:0] awcache,
    output logic [3:0] awlen,
    output logic [3:0] awqos,
    output logic [5:0] arid,
    output logic [5:0] awid,
```

```
output logic [5:0] wid,
output logic [31:0] wdata,
output logic [3:0] wstrb,

input logic b_arready,
input logic b_awready,
input logic b_bvalid,
input logic b_rvalid,
input logic b_wready,
input logic [1:0] b_bresp,
input logic [1:0] b_rresp,
input logic [31:0] b_rdata,

output logic b_arvalid,
output logic b_awvalid,
output logic b_bready,
output logic b_rready,
output logic b_wvalid,
output logic [2:0] b_arprot,
output logic [2:0] b_awprot,
output logic [31:0] b_araddr,
output logic [31:0] b_awaddr,
output logic [31:0] b_wdata,
output logic [3:0] b_wstrb
);
MemoryBus#(
    .DATA_WIDTH(24),
    .ADDRESS_WIDTH(32),
    .MASTER_ID_WIDTH(8)
) ps();
MemoryBus#(
    .DATA_WIDTH(24),
    .ADDRESS_WIDTH(32),
    .MASTER_ID_WIDTH(8)
) ray_m();
MemoryBus#(
    .DATA_WIDTH(24),
    .ADDRESS_WIDTH(32),
    .MASTER_ID_WIDTH(8)
) ray_s();
MemoryBus#(
    .DATA_WIDTH(24),
    .ADDRESS_WIDTH(32),
    .MASTER_ID_WIDTH(8)
) bram_bus();
MemoryBus#(
```

```
    .DATA_WIDTH(24),
    .ADDRESS_WIDTH(32),
    .MASTER_ID_WIDTH(8)
) dram_bus();

MemoryMaster mm(
    .clock(clock),
    .reset(reset),
    .in(gpio_in),
    .out(gpio_out),
    .bus(ps));
Crossbar xbar (
    .clk(clock),
    .rst(reset),
    .ray_m(ray_m),
    .ps(ps),
    .ray_s(ray_s),
    .bram(bram_bus),
    .dram(dram_bus)
);

RayTracer#(
    .POSITION_WIDTH(16),
    .DATA_WIDTH(24),
    .ADDRESS_WIDTH(32),
    .CONFIG_ADDRESS(0),
    .MASTER_ID_BASE(10)
) rt(
    .clock(clock),
    .reset(reset),
    .configPort(ray_s),
    .memoryPort(ray_m)
);

BusToAxiLite bram(
.clock(clock),
.reset(reset),

.bus(bram_bus),

.arready(b_arready),
.awready(b_awready),
.bvalid(b_bvalid),
.rvalid(b_rvalid),
.wready(b_wready),
.bresp(b_bresp),
```

```
.rresp(b_rresp),
.rdata(b_rdata),

.arvalid(b_arvalid),
.awvalid(b_awvalid),
.bready(b_bready),
.rready(b_rready),
.wvalid(b_wvalid),
.arprot(b_arprot),
.awprot(b_awprot),
.araddr(b_araddr),
.awaddr(b_awaddr),
.wdata(b_wdata),
.wstrb(b_wstrb)
);

BusToAxi dram(
.bus(dram_bus),

.arready(arready),
.awready(awready),
.bvalid(bvalid),
.rlast(rlast),
.rvalid(rvalid),
.wready(wready),
.bresp(bresp),
.rresp(rresp),
.bid(bid),
.rid(rid),
.rdata(rdata),

.arvalid(arvalid),
.awvalid(awvalid),
.bready(bready),
.rready(rready),
.wlast(wlast),
.wvalid(wvalid),
.arburst(arburst),
.arlock(arlock),
.arsize(arsize),
.awburst(awburst),
.awlock(awlock),
.awsize(awsize),
.arprot(arprot),
.awprot(awprot),
.araddr(araddr),
```

```
    .awaddr(awaddr),
    .arcache(arcache),
    .arlen(arlen),
    .arqos(arqos),
    .awcache(awcache),
    .awlen(awlen),
    .awqos(awqos),
    .arid(arid),
    .awid(awid),
    .wid(wid),
    .wdata(wdata),
    .wstrb(wstrb)
    );


endmodule
```

6.15 System_xbar_wrap.sv

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/05/2019 06:07:02 PM
// Design Name:
// Module Name: System_xbar_wrap
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module System_xbar_wrap(
    input wire clock,
    input wire reset,

    input wire [31:0] gpio_in,
```

```
output wire [31:0] gpio_out,

input wire d_arready,
input wire d_awready,
input wire d_bvalid,
input wire d_rlast,
input wire d_rvalid,
input wire d_wready,
input wire [1:0] d_bresp,
input wire [1:0] d_rresp,
input wire [5:0] d_bid,
input wire [5:0] d_rid,
input wire [31:0] d_rdata,

output wire d_arvalid,
output wire d_awvalid,
output wire d_bready,
output wire d_rready,
output wire d_wlast,
output wire d_wvalid,
output wire [1:0] d_arburst,
output wire [1:0] d_arlock,
output wire [2:0] d_arsize,
output wire [1:0] d_awburst,
output wire [1:0] d_awlock,
output wire [2:0] d_awsize,
output wire [2:0] d_arprot,
output wire [2:0] d_awprot,
output wire [31:0] d_araddr,
output wire [31:0] d_awaddr,
output wire [3:0] d_arcache,
output wire [3:0] d_arlen,
output wire [3:0] d_arqos,
output wire [3:0] d_awcache,
output wire [3:0] d_awlen,
output wire [3:0] d_awqos,
output wire [5:0] d_arid,
output wire [5:0] d_awid,
output wire [5:0] d_wid,
output wire [31:0] d_wdata,
output wire [3:0] d_wstrb,

input wire b_arready,
input wire b_awready,
input wire b_bvalid,
input wire b_rvalid,
```

```
input wire b_wready,
input wire [1:0] b_bresp,
input wire [1:0] b_rresp,
input wire [31:0] b_rdata,

output wire b_arvalid,
output wire b_awvalid,
output wire b_bready,
output wire b_rready,
output wire b_wvalid,
output wire [2:0] b_arprot,
output wire [2:0] b_awprot,
output wire [31:0] b_araddr,
output wire [31:0] b_awaddr,
output wire [31:0] b_wdata,
output wire [3:0] b_wstrb
);
System_xbar dut(
.clock(clock),
.reset(reset),

.gpio_in(gpio_in),
.gpio_out(gpio_out),

.arready(d_arready),
.awready(d_awready),
.bvalid(d_bvalid),
.rlast(d_rlast),
.rvalid(d_rvalid),
.wready(d_wready),
.bresp(d_bresp),
.rresp(d_rresp),
.bid(d_bid),
.rid(d_rid),
.rdata(d_rdata),

.arvalid(d_arvalid),
.awvalid(d_awvalid),
.bready(d_bready),
.rready(d_rready),
.wlast(d_wlast),
.wvalid(d_wvalid),
.arburst(d_arburst),
.arlock(d_arlock),
.arsize(d_arsize),
.awburst(d_awburst),
```

```
    .awlock(d_awlock),
    .awsize(d_awsize),
    .arprot(d_arprot),
    .awprot(d_awprot),
    .araddr(d_araddr),
    .awaddr(d_awaddr),
    .arcache(d_arcache),
    .arlen(d_arlen),
    .arqos(d_arqos),
    .awcache(d_awcache),
    .awlen(d_awlen),
    .awqos(d_awqos),
    .arid(d_arid),
    .awid(d_awid),
    .wid(d_wid),
    .wdata(d_wdata),
    .wstrb(d_wstrb),

    .b_arready(b_arready),
    .b_awready(b_awready),
    .b_bvalid(b_bvalid),
    .b_rvalid(b_rvalid),
    .b_wready(b_wready),
    .b_bresp(b_bresp),
    .b_rresp(b_rresp),
    .b_rdata(b_rdata),

    .b_arvalid(b_arvalid),
    .b_awvalid(b_awvalid),
    .b_bready(b_bready),
    .b_rready(b_rready),
    .b_wvalid(b_wvalid),
    .b_arprot(b_arprot),
    .b_awprot(b_awprot),
    .b_araddr(b_araddr),
    .b_awaddr(b_awaddr),
    .b_wdata(b_wdata),
    .b_wstrb(b_wstrb)
    );
endmodule
```