6.111 Proposal: FPGA Ray Tracer

Parker Huntington and Cece Chu

October 29, 2019

1 System Overview

Our system can be broken down into 6 modules/subsystems: the ray tracer module, the memory subsystem, the display module, the computer interface module, top level FSM, and the configuration port module (in roughly descending order of complexity). This section provides high-level functional descriptions of these 6 modules, and the following sections give some more details of the submodules that make up the more complicated modules.

1.1 Ray Tracer

This module is responsible for calculating the pixels to be displayed. It takes the camera angle and scene information from memory, generates "rays" (vectors), and has many ray tracing units which compute the pixels in parallel. It communicates with the memory subsystem to access scene information and store the resulting scene information. This will be the most computationally intensive unit and constitutes the critical path for how quickly we are able to render images, so efficiency will be key.

1.2 Memory

The memory subsystem is responsible for controlling the flow of data in and out of storage elements. Our system will be structured such that the masters (initiators of memory requests) are the Ray Tracer (including individual Ray Tracing Units and the Ray Generator), the Top Level FSM, and Computer Interface, and the slaves are a BRAM for storing the octree data structure, the Config register, DRAM for storing frame buffers, and the Display Module's configuration register. We chose to store the octree structure in BRAM because it requires frequent access by the RTUs which constitute the critical path in the system, so we want it to be fast. The frame buffer is stored in DRAM mostly because of its size.

Since there are multiple masters operating independently that may want to simultaneously access data and also since the address space is split between three different slaves, various arbiters and other components are necessary to avoid bus contention. These submodules, as well as a description of the memory access protocol, are further described in the Memory Details section.

We anticipate that memory may end up being a bottleneck in the system. On the DRAM side, the display must be able to read out a frame of pixels every 30Hz for the display to

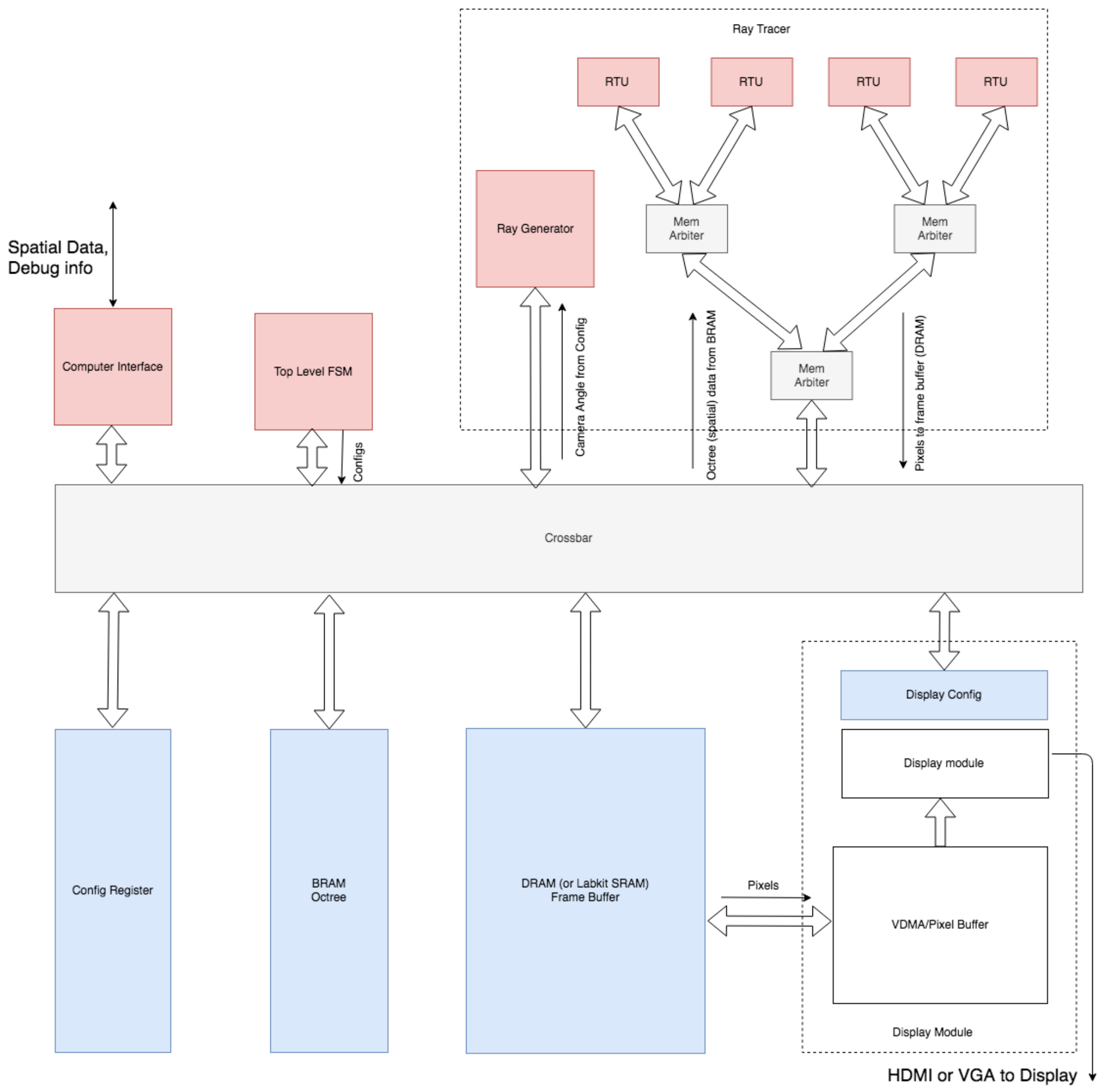

Figure 1: High-level diagram of various modules and how they are connected via the memory subsytem. Masters are in red, slaves are in blue, and the various arbitrators are in gray. Only 4 ray tracing units are shown for simplicity but more may be included in the final design depending on space; in this case we would just expand the binary tree of arbiters.

work properly. We think that the PYNQ Z2 board, which has a DRAM with a 16-bit bus @ 1050Mbps will provide sufficient bandwidth, but if not then we will use the labkit and store the frame buffers on the SRAMs instead.

1.3 Display Module

The Display module will be responsible for getting the pixels from the frame buffer(s) in memory and generating the appropriate video signals. Since we are displaying static images a lower refresh rate of 30Hz will help save bandwidth. It will also have a configuration submodule which acts as a slave on the memory bus and can be written to to change which frame buffer is being displayed. If we are successful at using the PYNQ board, we will use a VDMA module to get the pixels and output 720p HDMI with some VGA and VGA to HDMI converter IPs. If we end up using the labkit, we will have a pixel buffer submodule which repeatedly requests the next pixel from the frame buffer in DRAM to keep a pixel FIFO full. A VGA module based on the XVGA module from the labs will run on a 65MHz clock and generate hsync, vsync, hcount, vcount, and blank signals and get a new pixel value from the FIFO each time hcount or vcount is incremented. The FIFO is necessary to prevent arbitration between the Display Module and Ray Tracing from causing latency issues with the display.

1.4 Computer Interface

The computer interface provides a UART connection to the memory bus so that the memory can be initialized and config registers written to. This has the possibility of either being replaced with a soft processor or module that initializes it from the bus. If the Pynz Z2 board is used, then then the computer interface may serve as a bridge from the arm cpus without the use of the UART interface.

1.5 Top Level FSM

The top level FSM is responsible for setting the configurations of the ray tracer and display. It will have states for startup (writing the spatial information into DRAM), render (the frame is being rendered by the ray tracer), and display (the frame is done being rendered and is ready to be read by the display module). If we want to have multiple frame buffers for creating animation, the switching of the frame buffers will also be done by the top level FSM by swapping the frame buffer addresses for the display and the ray tracer.

1.6 Configuration Port

The configuration port is a slave on the memory bus that configures the ray tracer and probes the status. See Table 1 for the preliminary memory mapping. It needs to be assigned 5-bits in the address space in order to hold all of the entries.

Offset	Value
0x00	Camera $\hat{q}$
0x03	Camera $\hat{v}$
0x06	Camera $\hat{x}$
0x09	Camera $\hat{y}$
0x0C	Frame address
0x0D	Frame width
0x0E	Frame height
0x0F	Scene address
0x10	Status (1 = Busy/Start)

Table 1: The offset of the base address is given for each of the configuration values. The first four listed are 3-vectors, and thus take up multiple addresses.

2 Ray Tracer Details

The ray tracer is responsible for the actual computations in the system and doesn't contain any of the memory (ie. for frame buffer and rendering). The high level flow through the system is relatively straight forwards. The system is set up through the configuration port and a frame draw is started. This causes the ray generator to calculate rays corresponding to each pixel of the frame, and attach them to the memory addresses that each pixel will be put in. A ray unit is then assigned to the ray by the RU ray bus, which carries out the computation for that pixel. After it is finished, the pixel is sent out on the memory bus to be written into the frame buffer.

Due to uncertainty about the output pixel bit depth (12-bit, 16-bit, or 24-bit) and DRAM performance, the frame buffer structure can't be well predicted. This may mean that the frame buffer won't reside on the memory bus due to needing more bits to achieve a 24-bit output, although 24-bit output would most likely be achieved by dithering, and thus avoid this issue.

The ray tracing module will be developed and tested primarily through verilator. This will give us a robust understanding of the memory requirements and allow us to better plan and optimize the memory.

2.1 Ray Unit

A ray unit is responsible for taking a ray corresponding to a particular pixel, calculating the pixel color based off of the raytracing approach, then writing that value into the frame buffer. Each ray unit will be coordinated by a major FSM module. This module will store the target address for the pixel, then latch the incoming ray information. The ray position is then used by the memory module to traverse the octree and locate the corresponding leaf node. The leaf node information can then be read out back into the major FSM. Based off of the leaf node information, the FSM can either choose to scatter the ray, or to propagate the ray using the ray propagation module. This is repeated until the color module determines that the color is sufficiently determined, the ray terminates, or the number of reflections reaches a maximum value. Finally, the color is written out from the color module to the

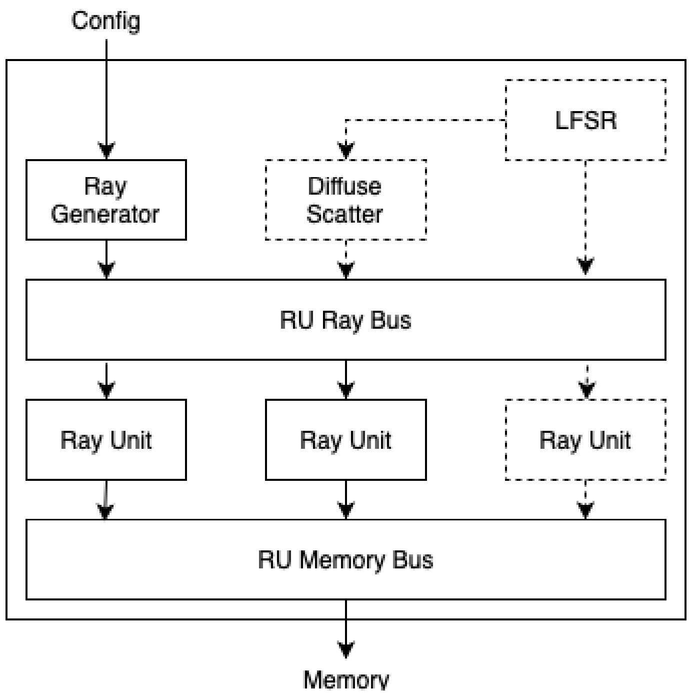

Figure 2: Ray Tracer block diagram. The LFSR and Diffuses scatter modules are dashed to indicate that they may not be present in the final design.

frame buffer.

2.2 Ray Propagation

The ray propagation module of the ray unit is responsible for taking a ray inside of some leaf axis aligned bounding box (AABB), then propagating that ray until it is just outside of the AABB. This process can then be repeated with the next leaf node to move the beam through space. In order to implement this in hardware, a binary search algorithm is used to find the boundary.

When the module starts, the position of the ray is stored into a position accumulator. Next, the direction vector is added to the position and compared to the boundary conditions. If the new position fits, then the result is stored back into the accumulator. Regardless of whether the position was stored, the direction vector is divided by two. As long as the direction vector is sufficiently normalized, this will converge to the final value.

There is a problem, however, with this method. Due to the calculations happening in fixed point integers, the divide by two causes a truncation that can lead the accumulator to come up slightly short of the boundary conditions. This is solved by holding one extra binary decimal place on the number, so that it can be rounded up when doing the calculations.

2.3 Volumetric Scattering

The binary ray propagation has some neat exploitable properties that we can use to do volumetric scattering if an approximate single cycle square root is possible. In this case, I

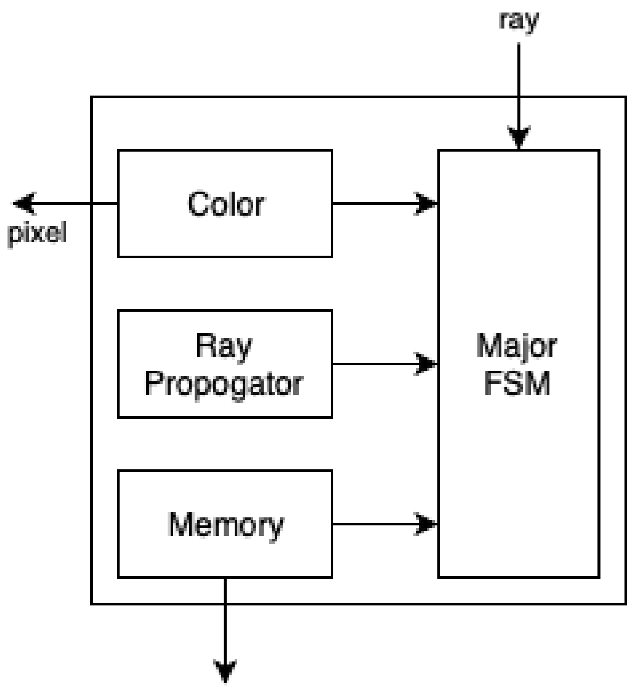

Figure 3: Shown above is the general structure of a ray unit.

take volumetric scattering to refer to a scattering where the scattering distance is a random exponential variable with uniform scattering cross section for simplicity. Due to hardware limitations, it is infeasible to calculate a random scattering distance using regular math operations.

When doing the binary search, at each iteration, we can ask if a scattering event happened within the interval of the direction vector. If the rate of scattering on the initial direction vector is λ, where each direction vector is normalized to length d then the zero scattering probability is

$$P_{zero} = e^{-\lambda t}.$$

This number can then be compared against a uniform pseudo-random number generated by a LFSR to determine if the scattering event occurred. If one is said to have occurred, then the new position isn't latched. A scattering event is then taken to have happened if the algorithm doesn't converge onto the boundary conditions as expected.

Each time we bisect the direction vector, since the length d is divided by 2, the P_zero factor simply needs to be square rooted. This, however, would require a decent square root approximation that operates in a single cycle.

3 Memory Details

3.1 Memory Protocol

Each master on the memory bus will have a 6-bit Master ID (MID) so the bus can identify destinations of data from read requests. In addition to the standard address and data lines, each memory access will also have master ID lines which either tell which master is initiating the request or the destination of the fetched data. When data is returned from one of the slaves, it will go to the data lines of all masters on the bus, but since the master ID is broadcast alongside the data only the master with the matching ID will know to process that data. Each slave will have to have a wrapper to keep track of master IDs of requests and "attach" them on data that is to be sent back.

We will use ready/valid handshaking protocol for performing memory accesses. When a master wants to perform a read or write request to one of the slaves, it will set its outgoing valid signal to 1 and wait to receive a ready signal. Once it receives a ready signal it will then have command of the address, data, master ID, and read/write lines and be able to perform the request. For read requests, once the slave finishes processing the request and is ready to return the data, it will set its outgoing valid signal to 1, wait to receive a ready signal, and then set the data and master ID lines with the appropriate values.

3.2 Memory Arbitrator

The memory arbitrator (Figure 4) is a binary arbitrator between which allows one of two masters to control the address and data busses when it receives a ready signal from downstream. If only valid1 or valid2 is high, it will assert the outgoing ready signal to the corresponding master when its input ready signal is asserted. If both valid1 and valid2 are asserted (two masters contending), it will arbitrate to the least-recently used (LRU) of the two, kept track of in a register. The logic inside will be combinational and in essence a large mux or muxes, so theoretically we will be able to construct a binary tree of arbiters without introducing extra cycles of latency for memory requests. As you can see on the right of Figure 4, data and master ID from the slaves is simply broadcast back to all masters.

3.3 Crossbar

The crossbar is the main interface between masters and slaves. It has a similar structure to the binary memory arbitrator, except that it has more than two masters and more than one slave. This means that it must also have logic to direct requests to different slaves based on the incoming addresses, arbitrate between more than two masters (also using a LRU method), and also arbitrate in the case of multiple slaves trying to simultaneously return data.

3.4 Memory Wrapper

The overall functionality of this module is to interface with the memory element and service read/write requests using our ready/valid protocol. For the PYNQ board, we would interface

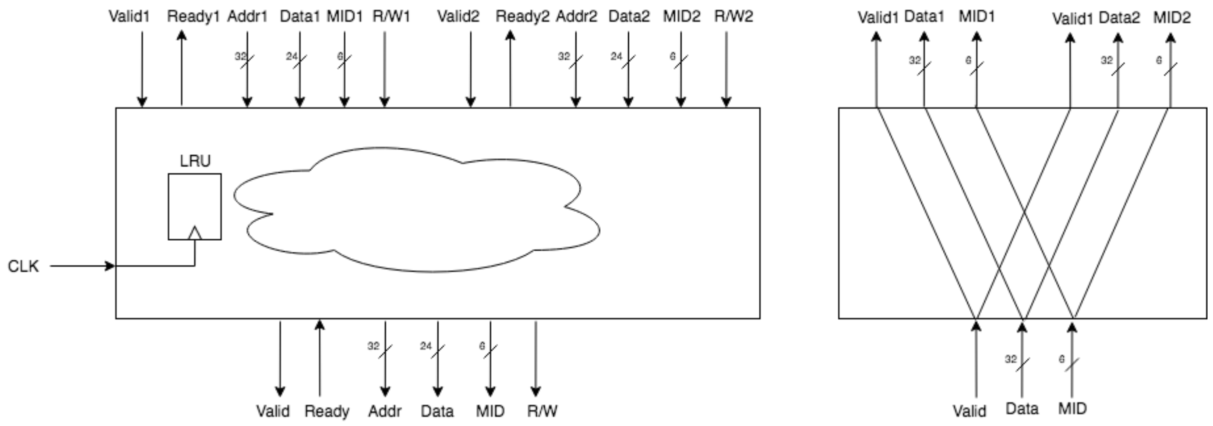

Figure 4: Block diagram of the memory arbitrator unit. This module arbitrates between two masters for read/write requests and passes data back to both masters for returning data from the memory.

with the memory controller through an AXI port on the built-in ARM microprocessor. For the labkit, we would create our own module to interface with the ZBT SRAM.

4 Division of Labor

Name	Area
Parker	Ray Tracer, Ray Units, Config Port
Cece	Memory subsystem, Display Module, Computer Interface