NTNU | Fakultet for informasjons-
teknologi og elektroteknikk
Institutt for elektroniske systemer

TFE 4295 Computer Design Project

TripSitter

Authors:
Gruppe 2
Peter Johan Flått-Bjørnstad
Henry Grindheim Hogstad
Markus Bøyum Johansen
Marcus Venner Hagberg
Daniel Yang Hansen
Ida-Sofie Pettersen
Kolbjørn Austreng
Sigvart Hovland

Trondheim, 2023

Contents

1 Introduction

TripSitter is a custom board composed of an EFM32 microcontroller and an Artix 7 FGPA. Its purpose is to render graphics to an external screen via an on-board VGA connector. To complete this task, the microcontroller acts as the brain of the board, keeping track of an updatable world consisting of spheres. The FPGA is leveraged to offload easily parallelizable operations from the microcontroller.

The rendering scheme that is implemented on TripSitter is a form of *ray tracing*. Ray tracing is a rendering scheme in which virtual *perceptive rays* are shot out from a camera onto a scene of geometric objects. Each ray is projected forward to ascertain information about the distance from the camera and objects that it hits, as well as physical properties of said object - such as color and reflectivity [1]. Ray tracing underpins most of modern high-end *computer generated imagery* as seen in animation movies or images where a high degree of realism is desired, but can be very computationally demanding for arbitrary shapes. For this reason, TripSitter focuses on sphere rendering, since ray intersection equations can be determined analytically for this geometric primitive, thereby greatly simplifying computational load.

Ray tracing is a method that lends itself nicely to the rendering of intricate materials and texture effects. A handful of these, namely *lambertian*, *metallic*, and *dielectric* materials were originally considered, but ultimately dropped due to project constraints. Rather, TripSitter concerns itself only with the rendering of *normal mapped* materials. These are materials where a surface's unit normal vector at any point is interpreted as a color triplet, thereby yielding a smooth gradient effect for smooth objects. The reader will find a discussion of the other aforementioned materials, and their potential implementation, under further work in section 7.

2 Sphere Rendering

As outlined in section 1, TripSitter is limited to sphere rendering. As such, theoretical background for a sphere based ray trace renderer will be presented. While the general priciples are valid for any ray tracer, our object intersection equations will be specific to spheres.

2.1 Sphere Geometry

A sphere can be uniquely defined by a point and a radius [2]. The point represents the center of the sphere, while all points that lie one radius away from this center, together define the surface of the sphere. In 3-dimensional Cartesian coordinates, we may describe this surface using 1, assuming the sphere in question is centered at origin

$$x^2 + y^2 + z^2 = r^2 \tag{1}$$

Shifting the sphere center to an arbitrary point $C = (C_x, C_y, C_z)$, eq. (1) now becomes eq. (2).

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2 \tag{2}$$

Defining two vectors, $\mathbf{p} = [x, y, z]^{\mathrm{T}}$ and $\mathbf{c} = [C_x, C_y, C_z]^{\mathrm{T}}$, it is possible to rewrite eq. (2) as eq. (3), where a superscript T denotes the transpose operation:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) = (\mathbf{p} - \mathbf{c})^{\mathrm{T}}(\mathbf{p} - \mathbf{c}) = r^2 \tag{3}$$

Next, parameterizing $\mathbf{p}$ as a the sum of an initial location, plus an arbitrary step in a given direction, we can express $\mathbf{p}$ as $\mathbf{p} = A + \mathbf{b} \cdot t$. Here, A represents a point (A_x, A_y, A_z), while t represents some step along a generic vector $\mathbf{b}$. With this representation of $\mathbf{p}$, we may now derive an analytical expression for the intersection between $\mathbf{p}$ and a sphere defined by eq. (3), as seen in eq. (4):

$$\Big((A + t \cdot \mathbf{b}) - C)\Big)^{\mathrm{T}} \Big((A + t \cdot \mathbf{b}) - C)\Big) = r^2 \tag{4}$$

Expanding and rearranging eq. (4), we obtain a second degree polynomial for the ray intersection distance, t, as seen in eq. (5):

$$t^2 \mathbf{b}^{\mathrm{T}}\mathbf{b} + 2t\mathbf{b} \cdot (A - C) + (A - C)^{\mathrm{T}}(A - C) - r^2 = 0 \tag{5}$$

Where we have used the following shortcut in notation: the points A and C may be interpreted as "pseudo vectors". That is, mathematically they behave the same way as a vector from origin

out to each point.

Applying the beloved quadratic formula to eq. (5), we may express the ray intersection distance analytically as in eq. (6):

$$
\begin{aligned}
t &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\
&\text{where} \\
a &= \mathbf{b}^{\mathrm{T}}\mathbf{b} \\
b &= 2\mathbf{b} \cdot (A - C) \\
c &= (A - C)^{\mathrm{T}}(A - C) - r^2
\end{aligned}
\tag{6}
$$

This can be visualized geometrically as in fig. 1. The case where eq. (6) has two real roots corresponds to a ray hitting the surface of the sphere twice. The case of a single real root corresponds to a ray touching the surface in a single point. In the case where we have complex roots, the ray does not hit the sphere at all.

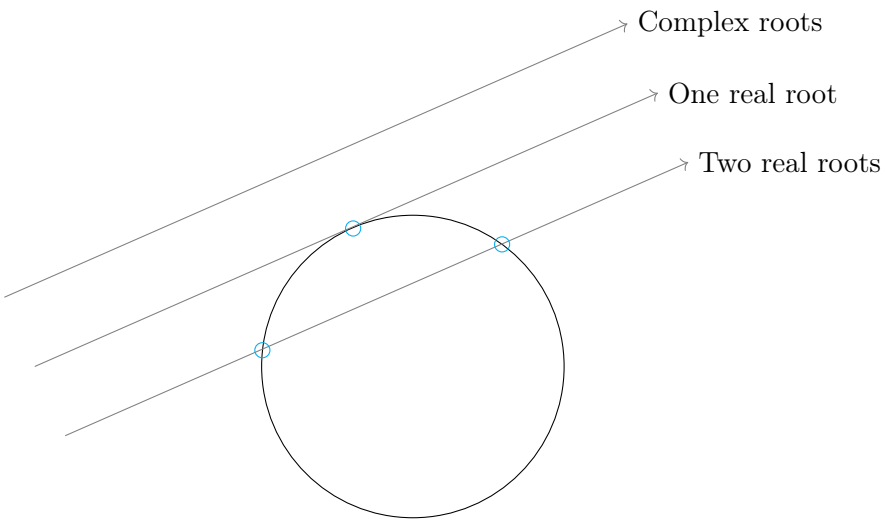


Figure 1: Sphere with different ray intersections.

2.2 Sphere Properties

Under the ray tracing paradigm, sphere properties such as materials and texture colors can easily be implemented by following two principles:

1. Each sphere has a base color contribution, that it will add to each incident ray. This effectively models the concept of *albedo*, namely a physical object's ability to reflect light. By defining a three dimensional albedo, one can succinctly account for materials reflecting different wavelengths differently.
2. Each sphere also has *reflective* and *refractive* constants, with differing probability distributions giving rise to different materials. Reflective materials will tend to reflect inbound rays, after adding their color contribution. If the reflective probability distribution is close to uniform, the material will look metallic. If a material refracts more than it reflects, that material will tend to look glass like, or crystal like.

Taking the base color of a material, along with its reflective and refractive properties, one effectively has a set of knobs that may be tweaked to create a wide range of realistic looking textures. By introducing slight variations in how rays are reflected or refracted, matte and cloudy textures can also be supported with very little additional complexity.

3 Implementation

3.1 PCB Schematics

The PCB Schematics are organized hierarchically, mainly into 3 modules:

- The Microcontroller and its peripheral units
- The FPGA and its peripheral units
- The Power System

... which is shown more detailed in *Figure 2*:

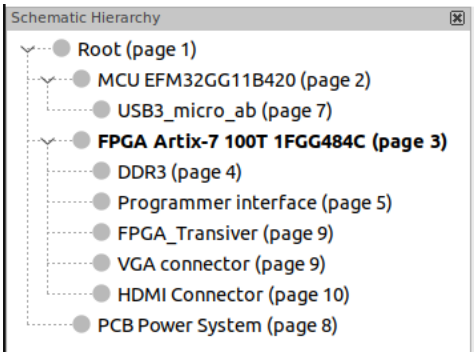


Figure 2: Hierarchical organization of KiCAD Schematic Sheets.

Overall, the main communication between the MCU and FPGA happens over QSPI. For this bus, the MCU operates as the bus master, with its own SRAM and the FPGA as the two chip selectable QSPI slaves. This bus is the main part of the information pipeline, so it is vital that its signal operates on a frequency that both the MCU and FPGA are compatible with.

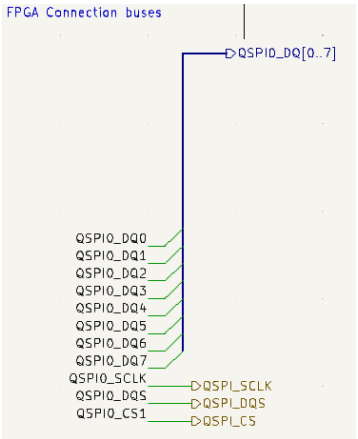
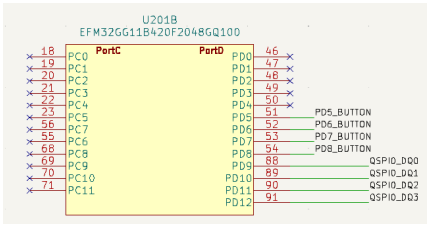
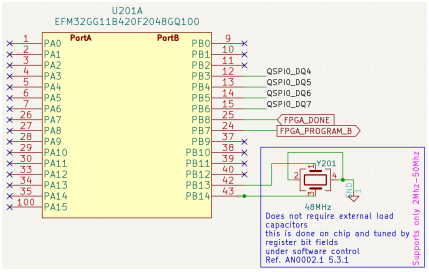


Figure 3: QSPI Bus signal format.

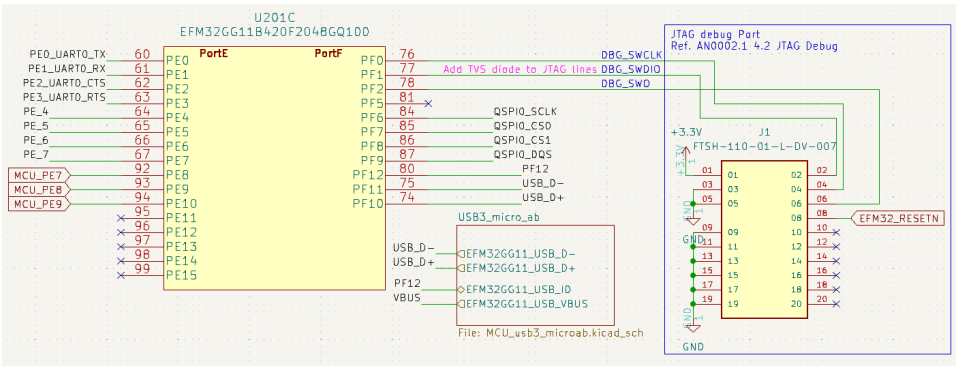
3.1.1 Schematics - Surrounding the Microcontroller



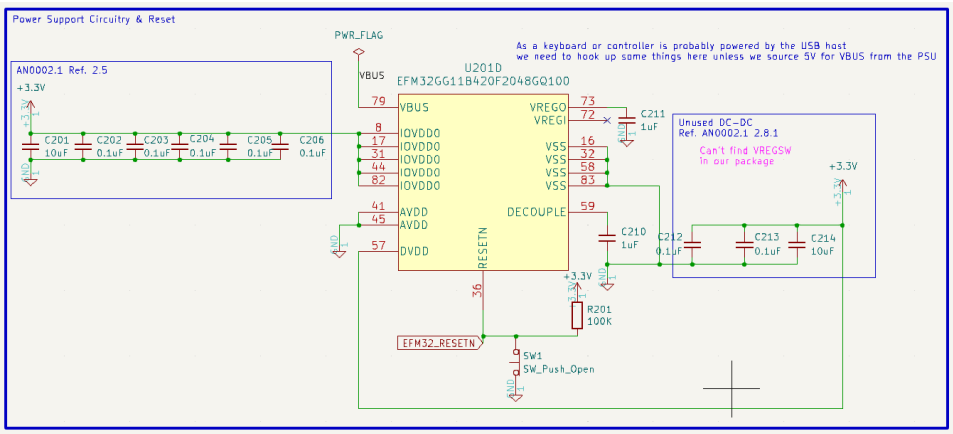
(a) MCU pins routed to QSPI Data Signals and I/O buttons.



(b) MCU pins routed to QSPI Data Signals and Oscillator.

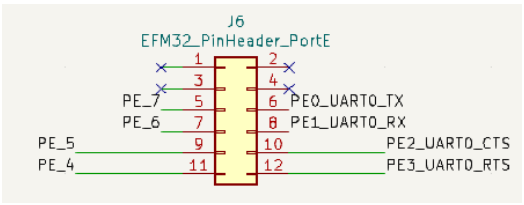


(c) MCU routing for QSPI Control Signals, USB, and JTAG debug port pin header.

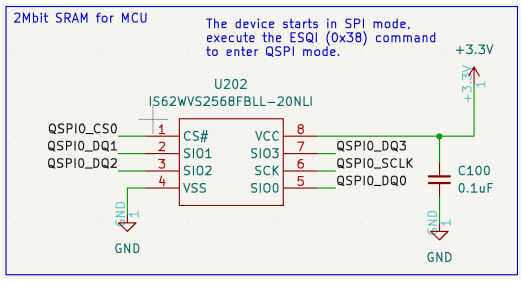


(d) Power Support Circuitry for MCU. Need to supply 3.3V to MCU, and 5V to power the USB connector. Includes a pulldown connector switch for short circuiting the reset signal.

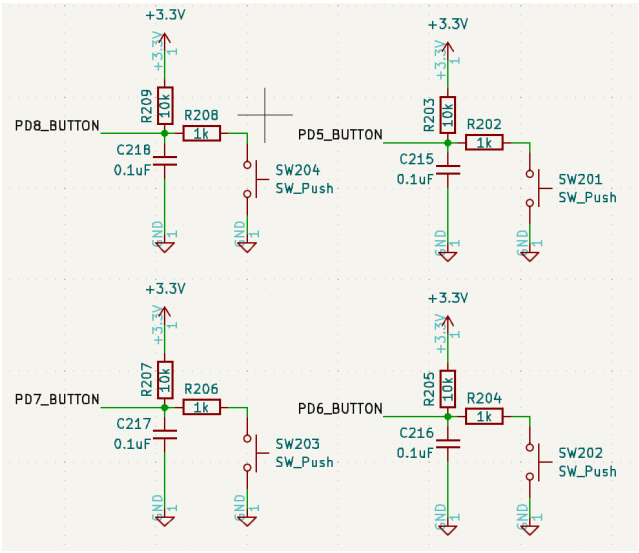
Figure 4: Schematics related to the microcontroller



(A) Pin Header for debugging.



(B) Dedicated 2Mbit SRAM for the MCU. Uses the same QSPI master as the one used to communicate from MCU to FPGA.



(C) MCU Button input. Primarily meant as a backup if USB keyboard interaction fails.

Figure 5: Additional schematics related to the microcontroller.

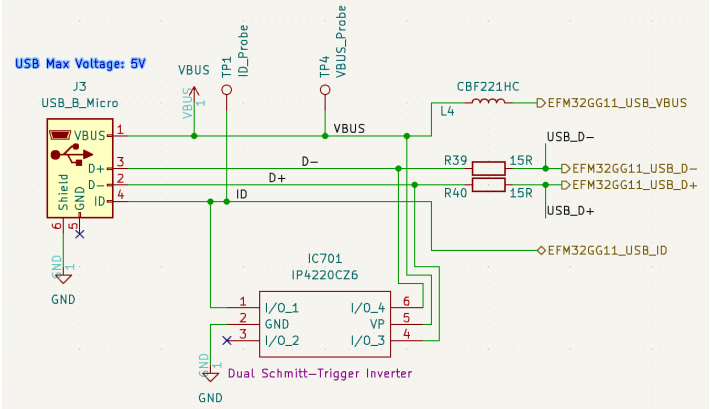


Figure 6: USB Connector Circuit. Dual Schmitt-Trigger Inverter used as surge protection.

3.1.2 Schematics - Surrounding the FPGA

The the FPGA has most of the computation responsibilities, so it is connected to many different components. While the FPGA pins are mostly all-purpose GPIO pins, we've tried to organize all signal buses logically. Notably, as show in *Figure 7*, the FPGA is separated into several I/O banks, with varying memory access latencies[1].

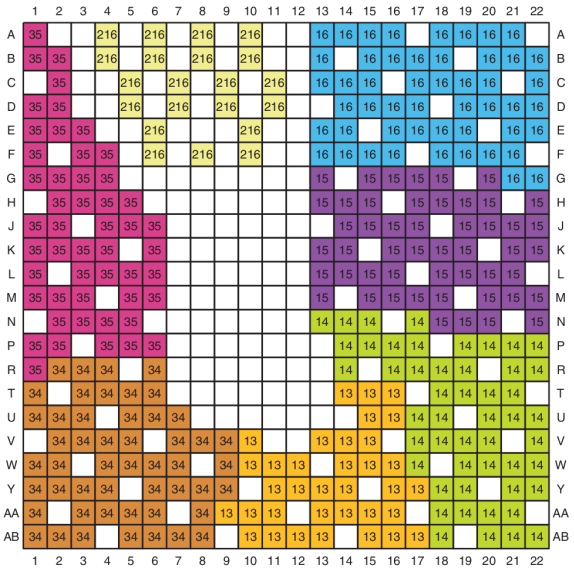


Figure 7: Pinout diagram for FGG484 Package XC7A100T, with respect to I/O banks. Uncolored areas are reserved for power signals and dedicated control signals.

This is how the different buses are currently spread across the different I/O banks:

- Bank 13: *unused*
- Bank 14: VGA Connector and Flash Programmer Interface and JTAG Debug signal. See *Figure 8*.
- Bank 15: *unused*
- Bank 16: HDMI Connector, QSPI Slave & interrupt signals from MCU. See *Figure 9*
- Bank 34: FPGA QSPI Master 1 & 2, and 2x4 Debug Header. See *Figure 10*
- Bank 35: *unused*
- Bank 216: *unused*

[1]Initially, it was intended to route DD3 DRAM to the GPIO pins in bank 34, something which proved to be impossible due to the physical design constraints given by the PCB manufacturer.

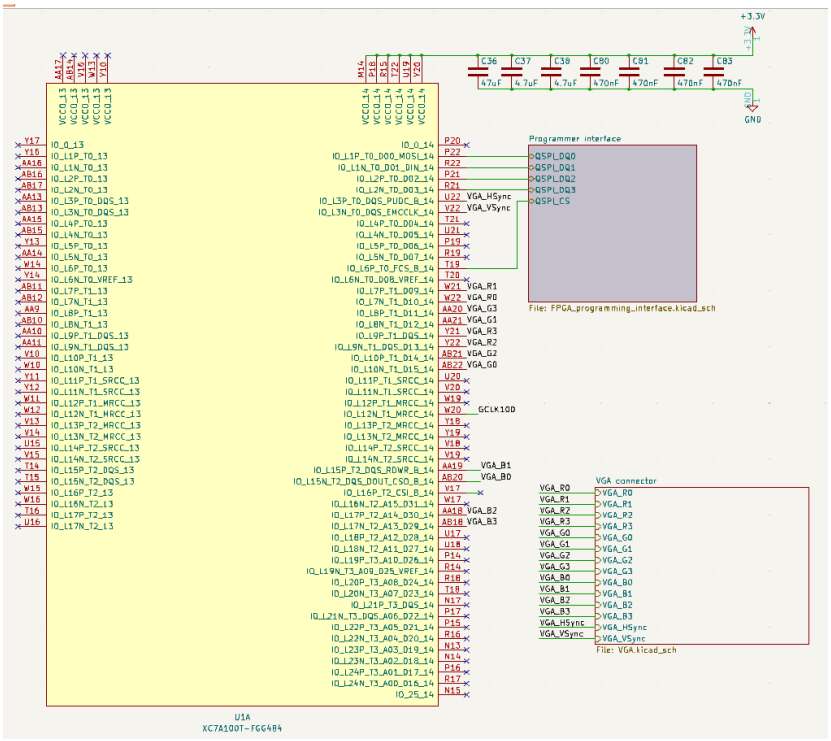


Figure 8: FPGA connections for I/O banks 13 and 14. Bank 13 power is disconnected.

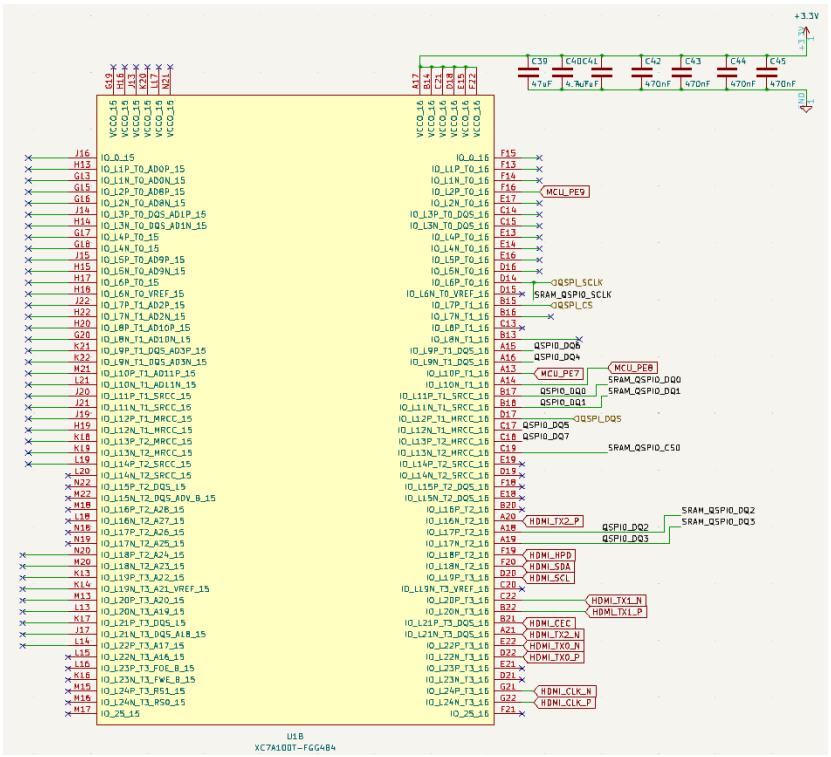


Figure 9: FPGA connections for I/O banks 15 and 16. Bank 15 power is disconnected.

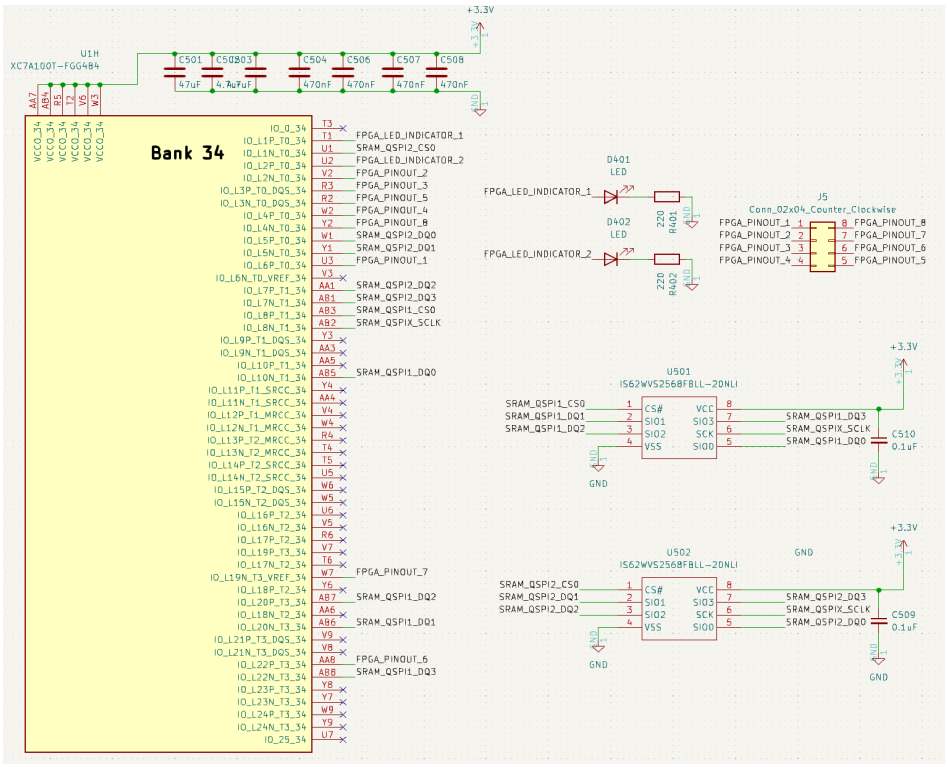


Figure 10: FPGA connections for I/O banks 34. Was originally going to be used for DDR3 DRAM, but was repurposed for SRAM, LEDs and debug pins.

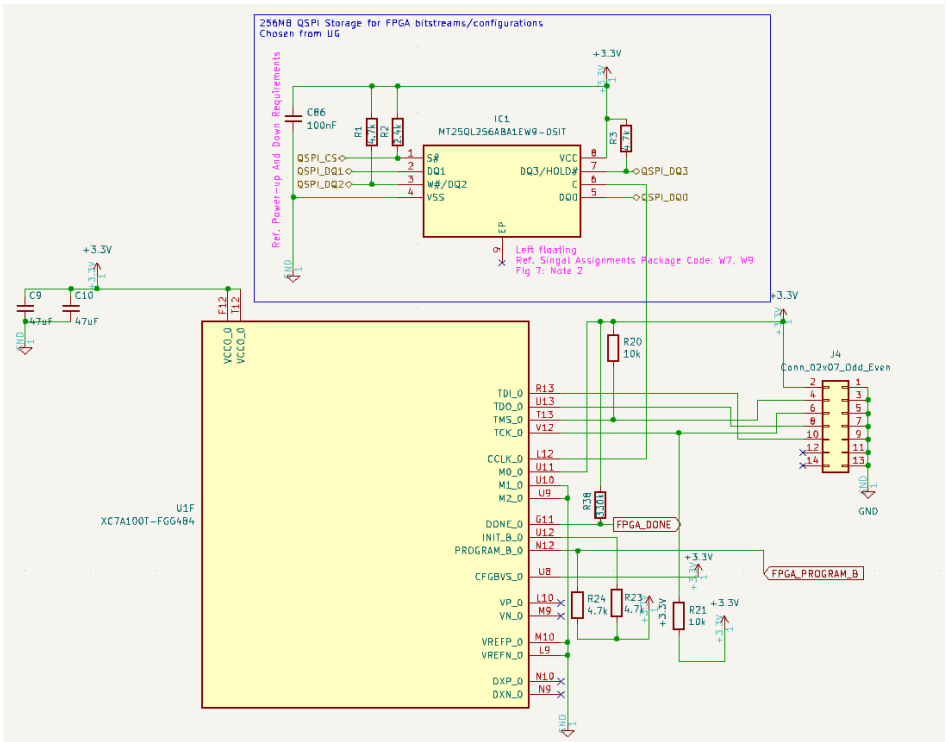


Figure 11: FPGA connections for control signals, debug header, and Flash Storage.

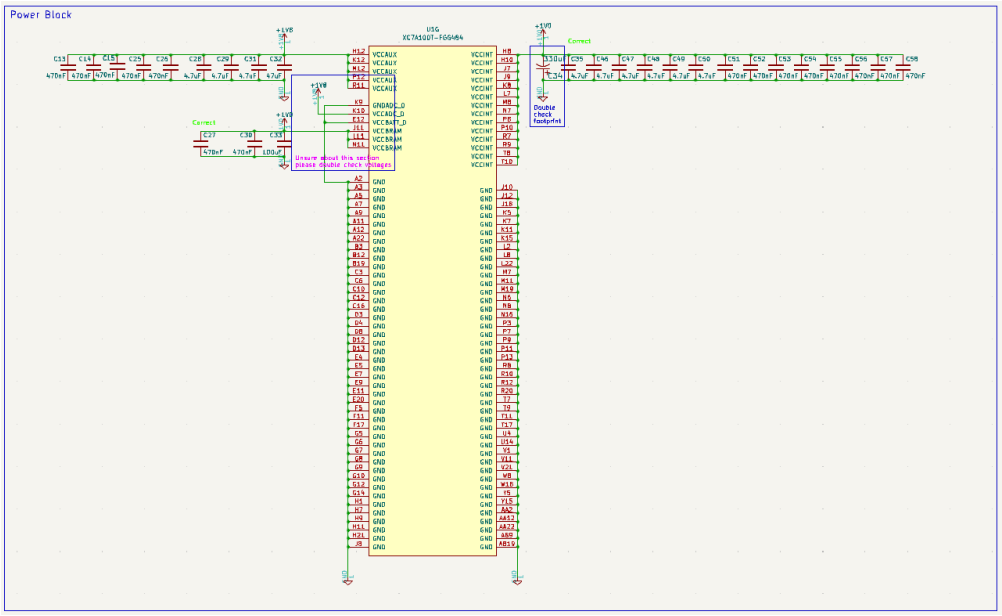
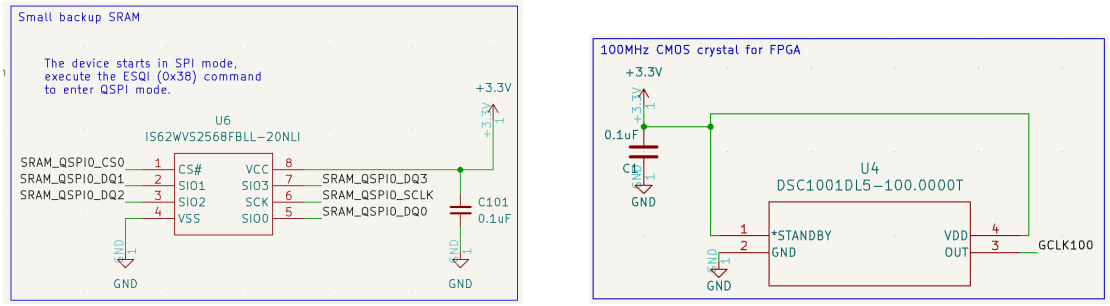


Figure 12: Power Block for FPGA. Sets up 1.8V and 1.0V power supply.



(A) Small 256Mb backup SRAM for the FPGA. Is routed to pick up on all signals sent through QSPI from MCU to FPGA. Ended up not using this backup in the final implementation.

(B) 100Mhz External CMOS Crystal Oscillator for the FPGA.

Figure 13: Schematic related to the FPGA

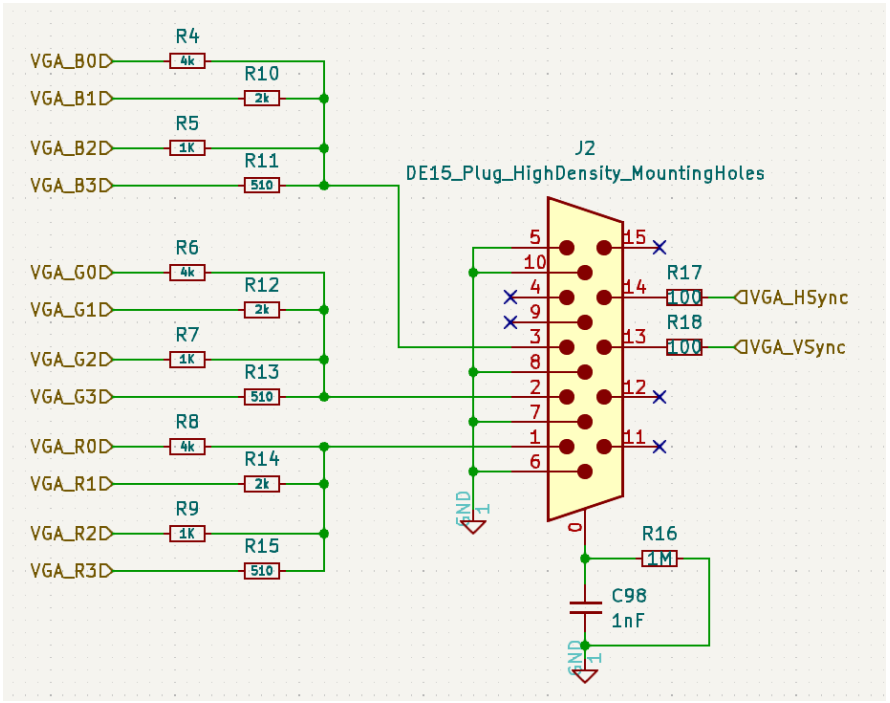


Figure 14: Support Circuit for VGA Connector.

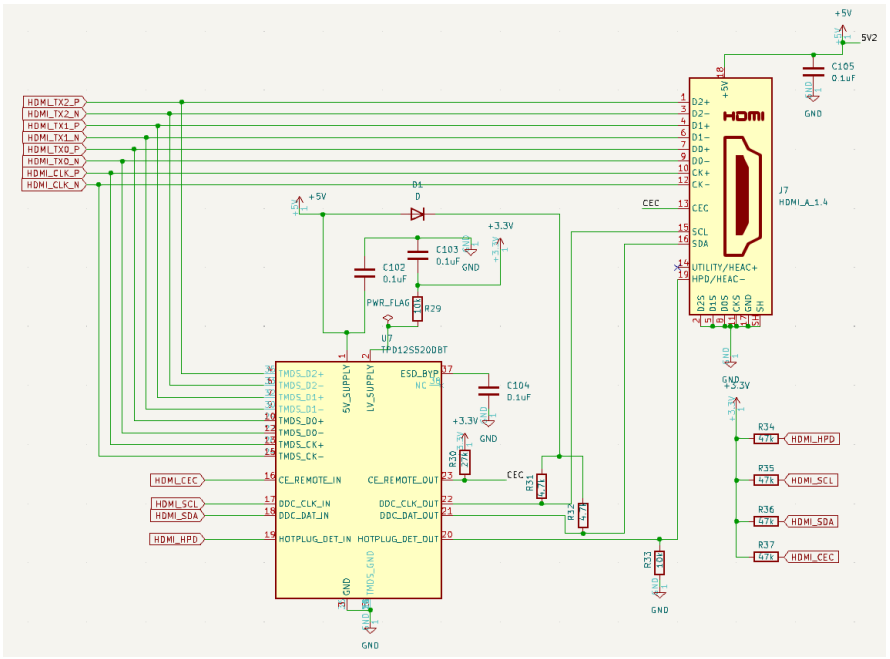


Figure 15: Support Circuit for HDMI Connector. As HDMI requires 5.0V, the support circuit performs logic leveling to/from 3.3V.

3.1.3 *Schematics - Power System*

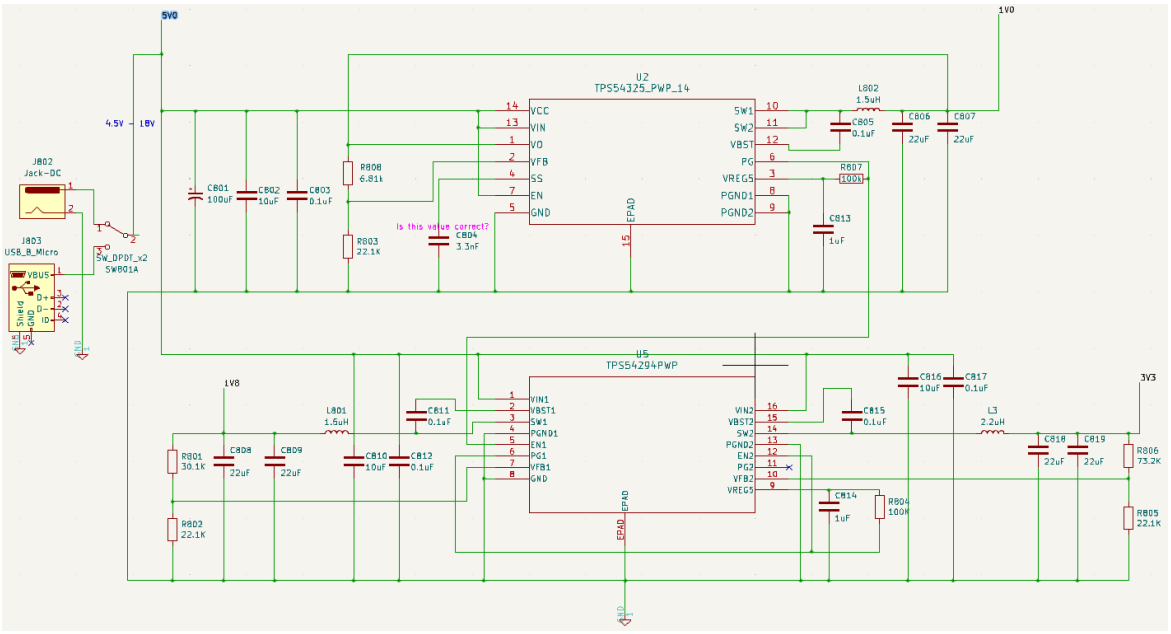


Figure 16: Power supply circuit system. Takes a 5V barrel jack or 5V Micro-USB as input, which is toggleable through a switch. Produces 4 different output voltages (in order): 1.0V, 5.0V, 1.8V and 3.3V.

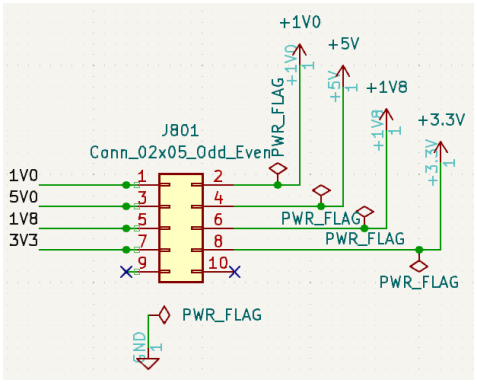
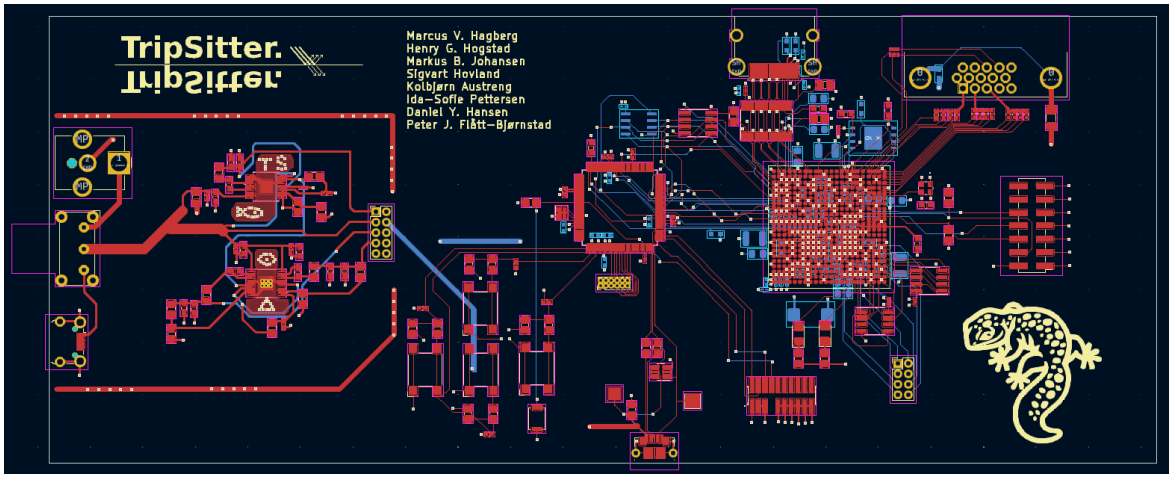


Figure 17: All of the power signals are routed through a short hat enable pin header, so that it is also possible to manually power the circuit from an external source.

3.2 PCB Layout



The PCB is divided into 4 layers: Front Signal Layer, Ground Layer, Power Inner Layer, Bottom Signal Layer.

Most of the components are soldered onto the Front Signal Layer, whereas the Bottom Signal Layer has been used mostly for resistors and capacitors. This is espescially the case for all decoupling capacitors and resistors used for the FPGA, due to the spatial constraint in the pitch between the FPGA pin connectors. Therefore, these components have been placed directly underneath the FPGA, as seen in *Figure 18*.

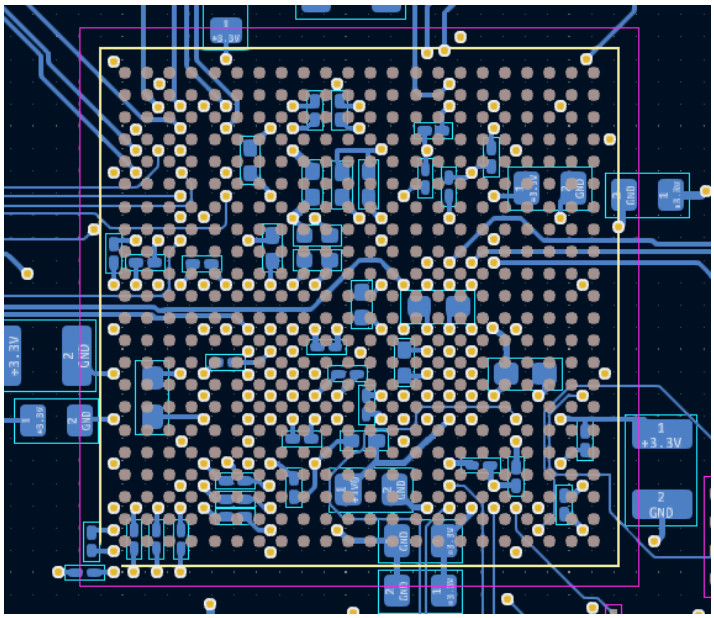


Figure 18: Capacitors and resistors are placed directly underneath the FPGA in order to adhere to spatial constraints.

As mentioned in section 3.1, we chose to not include the backup SRAM for the FPGA. However, the footprint for it has remained on the layout. Due to time crunch, this footprint was not removed. This has not caused any functional inconveniences, as the backup SRAM was placed on top of the existing QSPI slave bus from the MCU the FPGA (*Figure 19*).

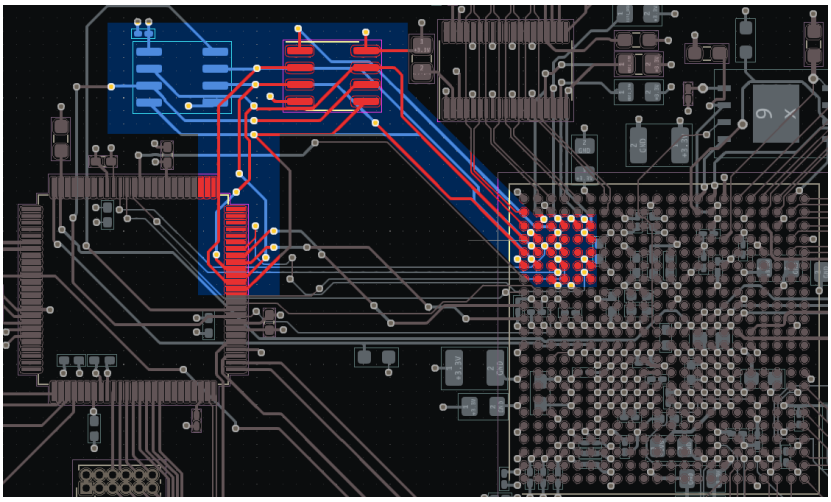


Figure 19: Highlighted: QSPI Bus with MCU as master. Slaves are SRAM (shown in blue, soldered on the Bottom Signal Layer), and the FPGA, which is also routed through the footprint for the FPGA's backup SRAM (the red footprint)

Additionally, when routing the HDMI bus, we had to take the individual differential pair signals into consideration, meaning we had to length match the signals (*Figure 20*).

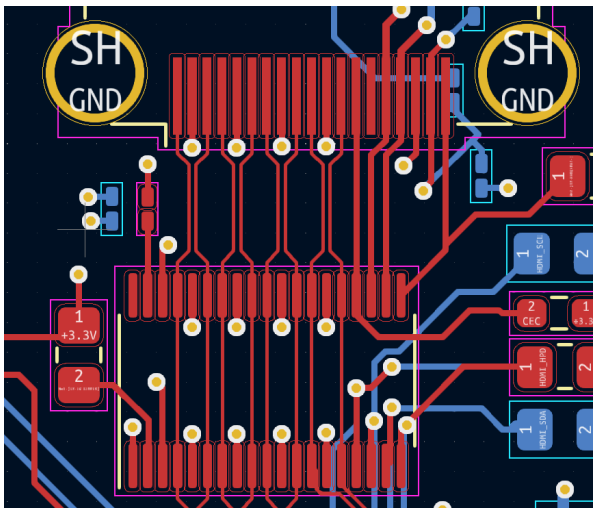


Figure 20: Length matched signals for HDMI input and support circuit.

3.3 The Finished PCB

While most of the PCB component are soldered correctly and working, some hot-fixes had to be made, and not everything worked as intended. A large hurdle throughout the soldering process was simply the lack of components. Several components were heavily delayed in shipping, leading to some "last minute" soldering work. A few components were also still held in customs at the time of the demo. This includes MCU's reset button, as well as all of the SRAM units. As can be seen in *Figure 21* and *Figure 22*, the 3 SRAM footprints (with 8 connector pads) on the front, and the 2 on the back are missing their corresponding components.

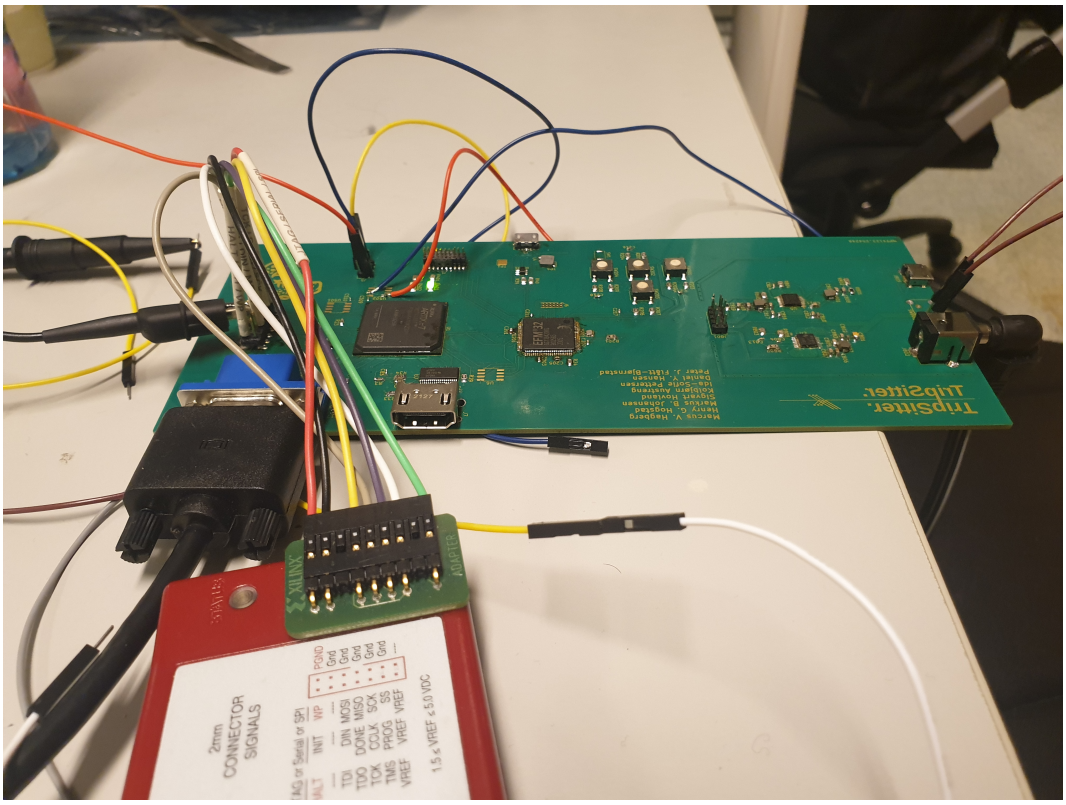


Figure 21: Finished PCB: Front Side. A lot of hardware hot-fixes have been made. The most noticeable adaptation is the array of cables connected to the red brick, which is for flashing the FPGA configuration.

A few of the components are visibly burned, which happened due an unnoticed prior reconfiguration of the solder oven. During the initial oven soldering process, the temperature was set too low, leading to the FPGA becoming only "half baked", with pin connectors on the underside of the FPGA not being connected properly. As this was discovered during a late troubleshooting, the board had to re-enter soldering oven with the rest of the components already mounted, causing some components to get slightly burnt, and others to detatch.

Lastly, the MCU was found to have a VDD pin connected incorrectly.

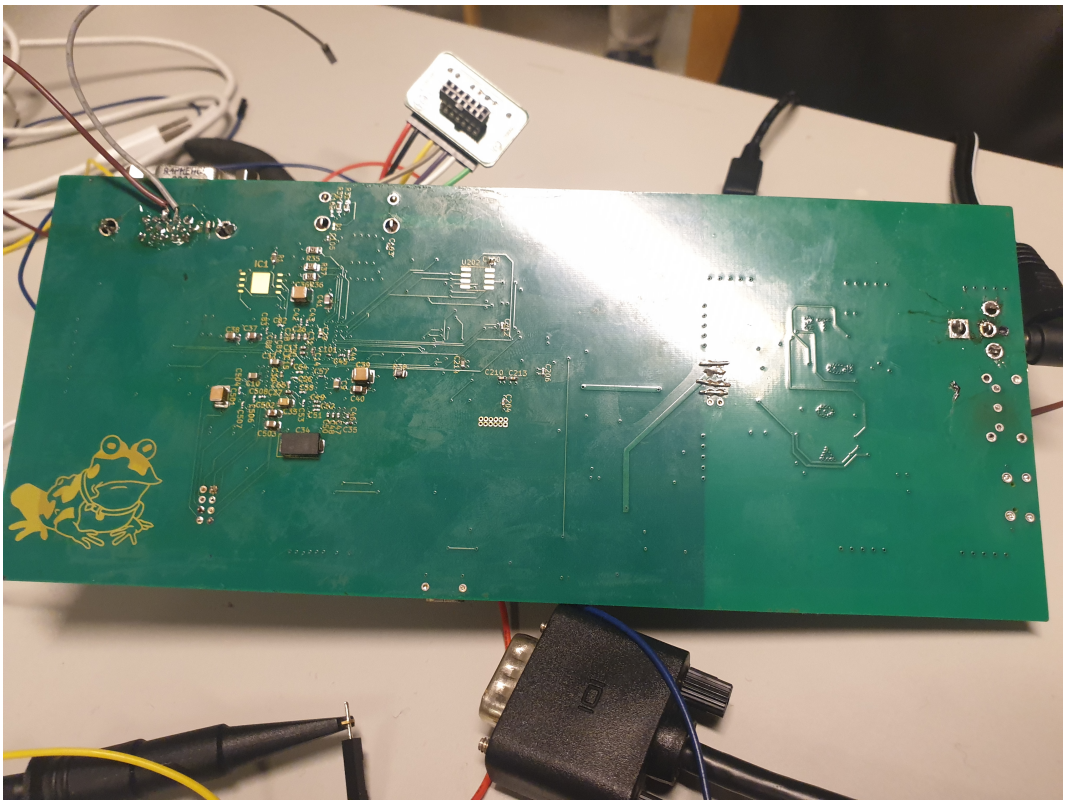


Figure 22: Finished PCB: Back Side.

3.3.1 *Various Hot-fixes*

Because of the aforementioned complications, several hot-fixes were made on the board. These can be seen in *Figure 23*, and include the following fixes:

- The form factor of the JTAG debug connector (J4) was too large, so many the pins were instead manually wired out to the red Xilinx brick on the left side of the image. This allowed the FPGA to be flashed directly over USB.
- As the local MCU was unable to function correctly, an external MCU dev board was connected through a JTAG debug pin header (J1).
- As the reset button never arrived, a "wire button" was facilitated instead, which are the orange and blue cables on the right side of the board.
- As the power input selection switch was lost in the soldering oven, a wire was soldered on to permanently use a 5V barrel jack power input.
- While the VGA port and its periphery was laid out and originally soldered correctly, the HSYNC signal was missing. This is likely due to one of the resistors (R17) getting dislodged in the soldering oven. Because the VGA port is mounted with through-hole vias, a cable (see the purple cable) was soldered to the underside, and connected to one of the pins on the FPGA dedicated debug pinout (J5).

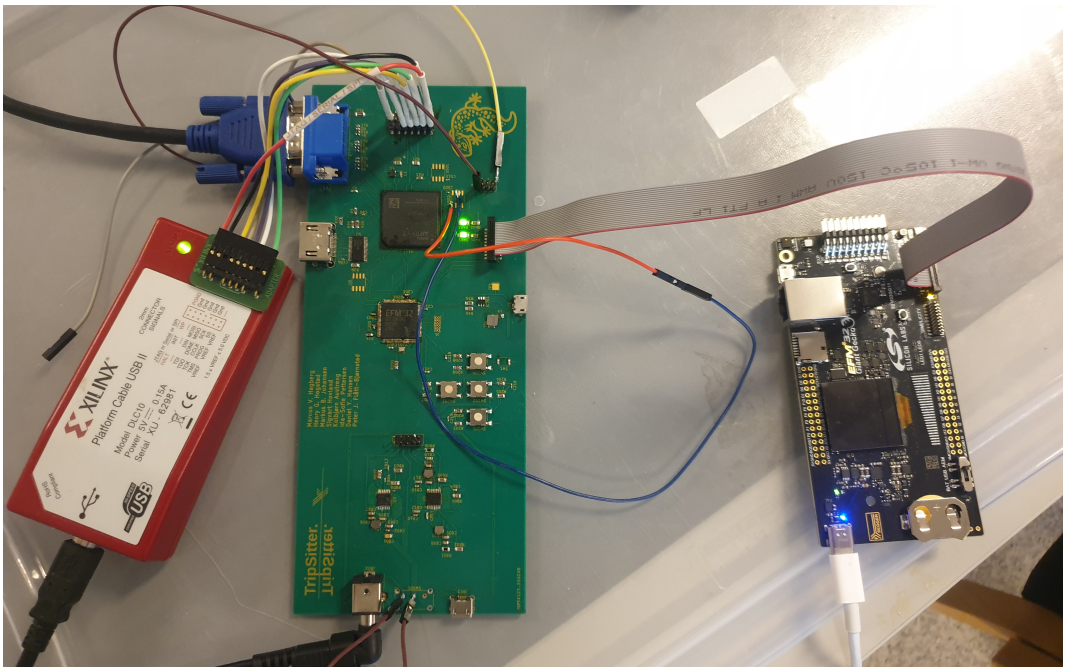


Figure 23: The PCB board set up to work. Because of complications with getting the MCU to work, the FPGA was instead connected directly to a MCU dev board.

3.4 Microcontroller

The Microcontroller (MCU) serves as the primary control unit for our ray tracing system. Key functionalities include:

- The MCU operates on floating-point values internally but transmits fixed-point representations to the FPGA to simplify computation on the FPGA side.
- User input is received through the Human Interface Device (HID) keyboard, connected over USB.
- Communicates with the FPGA via Serial Peripheral Interface (SPI) to transmit world data for rendering.
- Implements a camera system inspired by OpenGL, transforming coordinates from world space to view space using homogeneous 4x4 matrices for translation, rotation, and scaling in three dimensions.
- Maintains an updated internal world representation based on user input, sending the latest data to the FPGA over SPI.

3.4.1 USB Interface

The MCU functions as a USB Host, enabling communication with any keyboard adhering to the USB HID Keyboard standard. We use the em_usb.h library from Silicon Labs for the usb host stack.

See section 6.1 for keybindings.

3.4.2 SPI Communication

For SPI communication, we use the SPIDRV library from Silicon Labs. Upon completing a world update, the MCU synchronously sends the new data to the FPGA over SPI. To ensure seamless operation during data transmission and updating, two copies of the world are maintained. This approach guarantees a valid version is always ready for transmission, while the other copy can be modified without concerns about concurrent write operations during SPI transmission. The SPI-peripheral for the microcontroller puts a limit on how many bits can be sent during each transmission, meaning the data has to be deconstructucted before transmission, and reconstructed on the FPGA upon retrieval.

3.4.3 Camera System Implementation

Our camera system, drawing inspiration from OpenGL, initially defines the camera in world coordinates. Before rendering, a conversion to view space takes place, centering the camera at

the origin. The use of homogeneous 4x4 matrices facilitates affine transformations, leveraging matrix multiplication associativity. This efficient approach combines multiple transformations before applying them to each sphere's position (represented as vectors), optimizing performance on the MCU by limiting the number of matrix by vector multiplications and enabling smooth rendering on the FPGA by always keeping the center of the viewport at origin.

3.4.4 Fixed-Point Values

Internally, the MCU performs calculations using floating-point operations for enhanced precision. However, due to the absence of native floating-point support on the FPGA and the complexities of implementing such support, we convert these floating-point values to fixed-point representation before transmitting them over SPI.

While this conversion introduces a nominal computational overhead on the MCU, it simplifies the computational burden on the FPGA. Dealing with fixed-point representations on the FPGA is more straightforward, especially when native floating-point support is unavailable. The decision to use fixed-point values for communication facilitates seamless integration between the MCU and FPGA, striking a balance between computational efficiency on the MCU and ease of implementation on the FPGA.

3.5 FPGA

The FPGA in our system works as a *dumb* component - its only job is to calculate and visualizes the sphere sent by the microcontroller. Each sphere consists of its center-coordinates, radius and color. To emulate camera-movement, the spheres coordinates are updated so the difference between their original placement and their new position equals that of moving the camera.

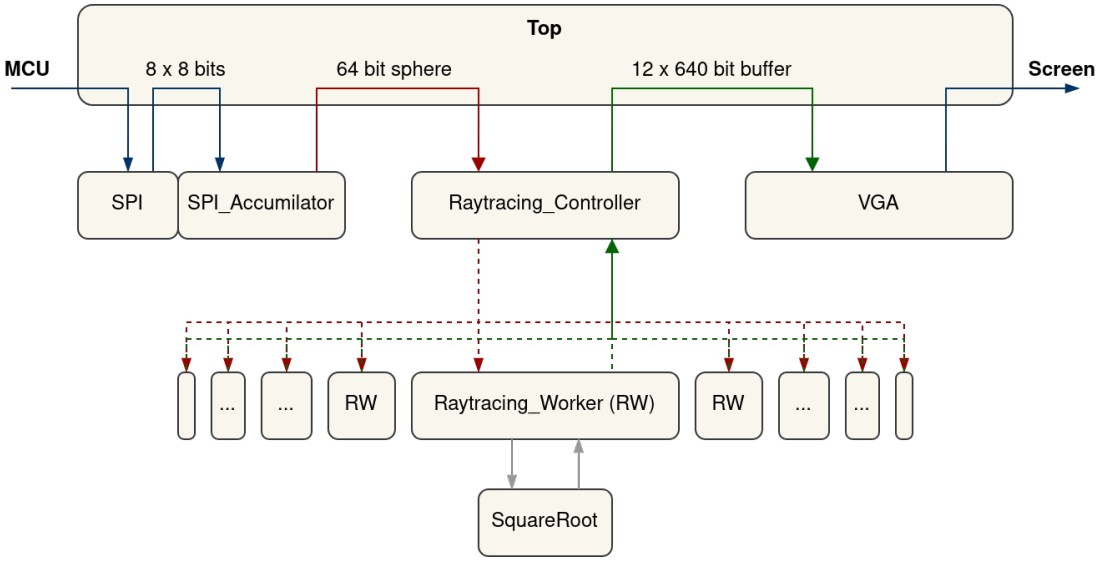
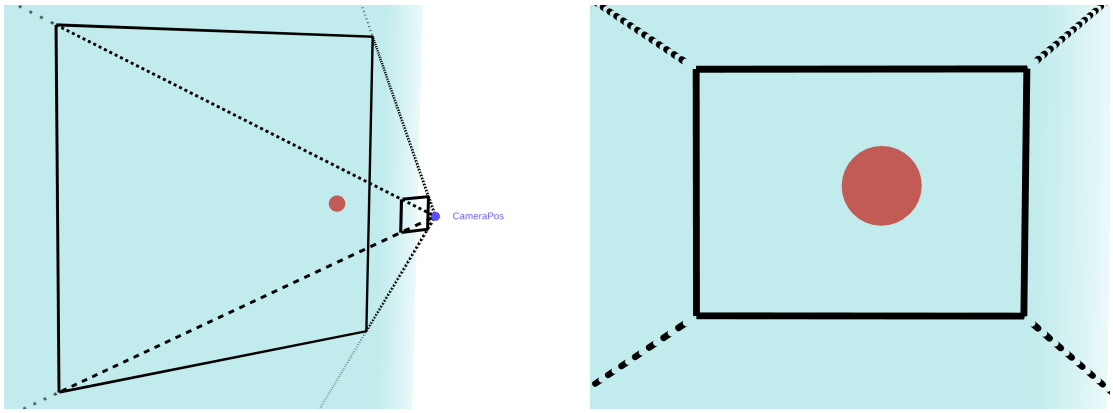


Figure 24: FPGA module flow from MCU, to the raytracing workers, and to the screen with VGA

Figure 24 illustrates the FPGA datapath. To render a sphere, the microcontroller (MCU) sends world-data through an SPI-interface. Raytracing_Controller instantiates the different Raytracing_Workers and keeps track of which workers are busy. The Top-module binds the different modules together and outputs the picture to a VGA controller. Figure 25a illustrates a sphere in the world as it is sent from the MCU, while fig. 25b shows the expected output on the screen.



(a) Placement of a sphere within the boundaries of our world.

(b) The same sphere viewed through the camera.

Figure 25: A sphere within our boundary values viewed in the world and through the camera.

3.5.1 VGA

To display the data, the FPGA sends color-values to a computer monitor using VGA. Each pixel is represented by a 12-bit value, i.e. 4 bits each for the red, green and blue intensities. For a standard VGA-resolution of 640 x 480 pixels, the pixels must be sent to the monitor at a frequency of 25 MHz, hereby referred to as the *pixel clock*. In addition to the pixel clock signal, two signals representing horizontal and vertical synchronization is sent to the screen, which infer where each pixel is positioned on the monitor. This work is done by the VGA module, and it also outputs some information about the current row being drawn to help determine which pixels are next to be calculated by the FPGA.

3.5.2 Raytracing Controller

To fully utilize the available resolution, one ray must be calculated for each pixel on the screen. Since the clock on the A7-100T is at 100MHz, it would mean there is about 4 cycles before the pixel clock needs the next color, which is impossible to calculate in time sequentially. Luckily, the calculation of each ray is independent, meaning they can be pipelined and/or be done in parallel. The final design used a parallel structure, as it was an easier way to divide programming-tasks within the team; one creating the controller which distributes and retrieves data from each parallel worker, the Raytracing Controller, and one implementing the calculations each worker has to do on that data.

3.5.3 Line buffering

The result of each worker must be stored in some buffer before it can be displayed on the monitor. The traditional way to do this is with two frame buffers, one written to by the parallel workers while the other is read by the VGA module. This becomes an issue when there is not sufficient memory available. To utilize the full range of 12-bit colors, a single frame buffer would have to be $12 \text{ bits} \cdot 640 \cdot 480 \approx 3686$ Kbits, using about 80% of the available memory on the A7-100T. This would require external memory, which in this case was either DDR3 or SRAM. It was unknown if the DDR3 would be implemented correctly, and the SRAM at 20Hz turned out to not be fast enough for the pixel clock. Instead of reducing memory by lowering the resolution or color depth, we went for reducing the size of the buffer. VGA draws one row at the time, from top to bottom, so instead of buffering two frames, one could buffer two rows, one read by the VGA and one for writing the next row. This line buffer would be the size of $12 \text{ bits} \cdot 640 = 7680$ bits, well within the available FPGA memory.

3.5.4 Multiple calculations for each worker

The next issue is how to calculate the 640 pixels of a row in parallel. The simplest way would be having one worker for each pixel in the row. This is feasible for small calculations, but in this case there is not enough available LUTs and DSPs for calculating 640 rays at the same time. The buffer would also be filled in about 14 clock cycles, underutilizing the $640 \cdot 4 = 2560$

available clock cycles. By reducing the number of workers and assigning each worker a set of pixels, it can strike a balance between available hardware and clock cycles.

3.5.5 Ray tracing Worker

To calculate whether a ray intersects with a sphere, and how far away it is, each worker implements the procedure outlined in section 2. The specific algorithmic steps are summarized in table 1 below. The majority of the steps taken in this algorithm are straight forward and self explanatory. The only exception being step 3, which involves taking the square root of an argument to determine the distance.

Calculating square roots using fixed point numbers on an FPGA will necessarily involve some finesse and a handful of approximations. This finesse has kindly been supplied to us by Zhongcheng Zhou and Jingchun Hu, through their paper `A Novel Square Root Algorithm and its FPGA Simulation`[3], and their willingness to share their VHDL code.

Table 1: Ray-calculation algorithm implemented in each worker.

1: Keep track of closes ray intersection	closest <= undefined
2: For each sphere:	
2.1: Square ray direction	$a = x_d^2 + y_d^2 + z_d^2$
2.2: Dot sphere-origin to ray-origin vector and direction	$b = 2 \cdot (r_x \cdot x_d + r_y \cdot y_d + r_z \cdot z_d)$
2.3: Determine if we are inside the sphere	$c = r_x^2 + r_y^2 + r_z^2 - radius^2$
2.4: Compute collision discriminant	$dis = b^2 - 4 \cdot a \cdot c$
3: Compute where we hit the sphere	if $dis < 0.0$: no hit - leave closest as is otherwise: $distance = -b - \frac{\sqrt{dis}}{2 \cdot a}$
4: Update closest	if closest is undefined or distance < closest $closest = distance$

4 How to use TripSitter

The github-project contains a few different bitstreams depending on whether you want to flash the PCB or the Devkit.

5 Setup - DevKit

The simplest setup is on the A-100T devkit without a microcontroller. Use vivado to flash *raytrace-devkit.bit*. Connect the VGA PMOD as shown in figure 4.

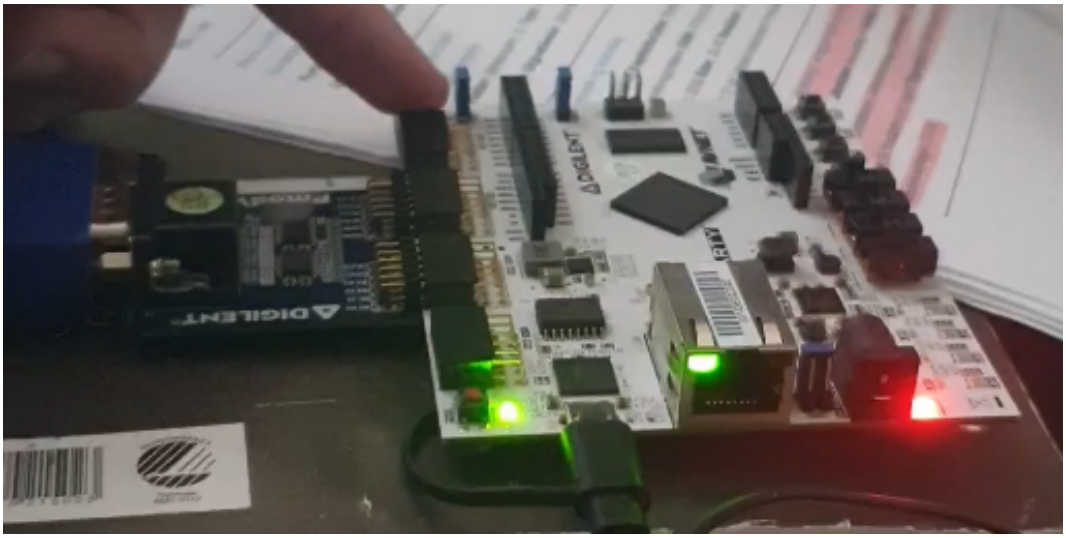


Figure 26: Setup of FPGA Devkit with VGA PMOD

This devkit can also communicate with a EFM32 devkit using the SPI-ports. A hex for the microcontroller is included in the project.

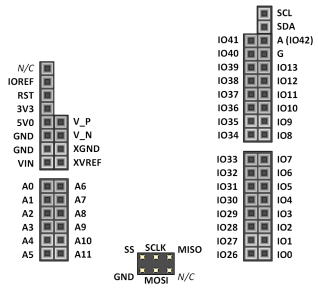


Figure 27: SPI pins on the fpga, we use MOSI, SS, SCLK and GND. Their equals are labeled in the MCU

6 Setup - PCB

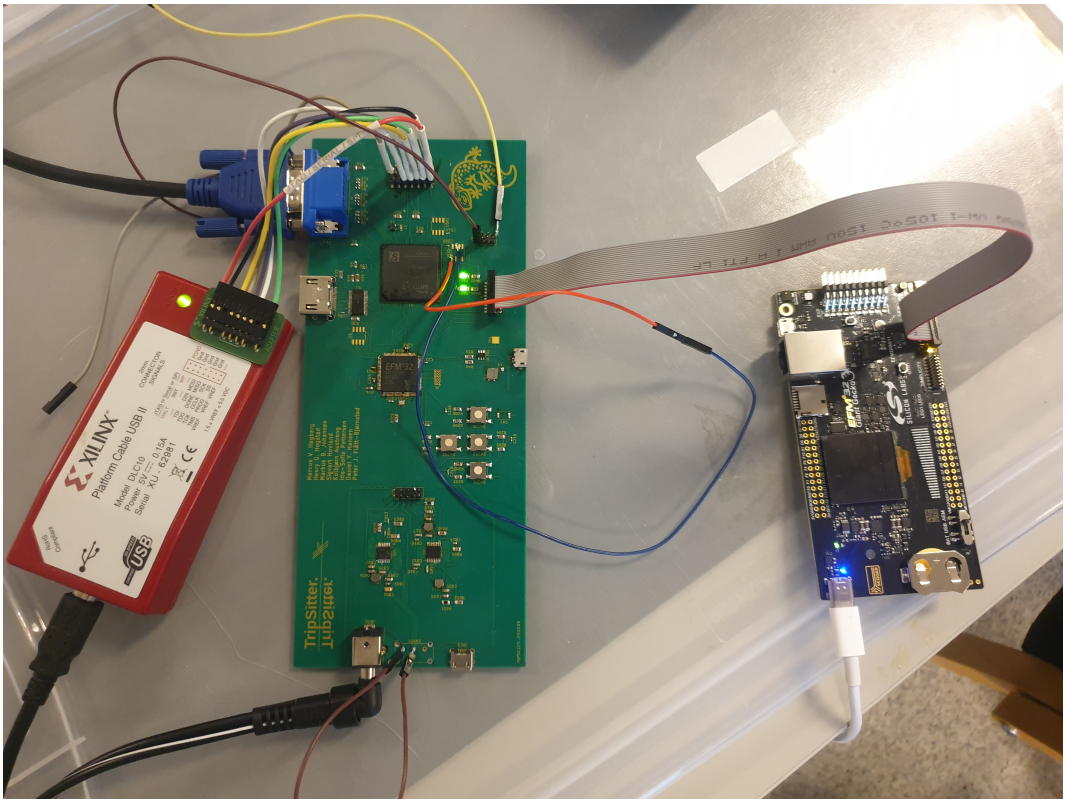


Figure 28: Setup for pcb. Purple cable connect the HSYNC of the VGA, while the red and blue cable resets the FPGA

The full use of the PCB requires connecting a JTAG Debugger to flash the FPGA, while the microcontroller on the PCB should be programmed to send data.

Flash the FPGA with *raytrace-pcb-working-with-spi-internal.bit* and connect power the VGA.

6.1 Keyboard controls

move the camera, use standard WASD buttons

To look around with the camera, use IJKL

To restart the MCU, click 'r'

7 Further work

As outlined in section 2, TripSitter is only capable of rendering a single sphere along with its normal map texture. The two most obvious avenues for further work thus are `1)` multiple spheres, and `2)` different material properties. We first outline the necessary high level rendering pipeline changes, before giving an overview of where these changes have to be accommodated.

7.1 High level changes

1: To support multiple spheres, the algorithm described is table 1 has to be repeated for each sphere in the scene. The algorithm definition itself already accounts for this. Hence, the primary change involves keeping more spheres in memory on the FPGA, and remembering which sphere intersection is closest to the camera viewport.

2: To cater to different textures and materials, the required changes are further reaching and more extensive. For all types of materials discussed in section 2, reflections must be supported. The simplest material to implement is *metallic*. This is a material for which there exists only reflections, no refractions, and each reflection is deterministically determined by the incident angle. This is what gives rise to the mirror like finish of polished metal.

3: Changes required in the rendering pipeline include multiple passes of rays. That is, on the first pass, rays are beamed out from the camera viewport as usual. Then, for each ray that has hit a reflective material, a new ray origin and direction is calculated based on the point of intersection, and the angle it makes to the surface. This potentially has to be repeated several times, due to rays bouncing between two or more reflective objects. This is generally unproblematic, as a cutoff value may be set, limiting the maximum number of bounces a ray can do. Color contributions would have to be mixed using a weighted average of all ray hits, necessitating extra book-keeping on the FGPA.

4: Building on the extensions introduced by metallic materials, *lambertian* materials are a next natural step. Lambertian materials are ideal matte materials that reflect incident rays based on a probability distribution rather than a deterministic angle-to-negative-angle relationship. A typical Lambertian reflection distribution is illustrated in fig. 29. Different renditions of matte materials may have alternate distributions than the one shown here, but they all require a source of randomization to pick the exact reflected direction. *Pseudo-random* numbers are well understood in the literature, and for an application such as TripSitter, a cheap and simple XOR-shift-based PRNG would suffice, such as the ones defined in `Numerical Recipes`[4].

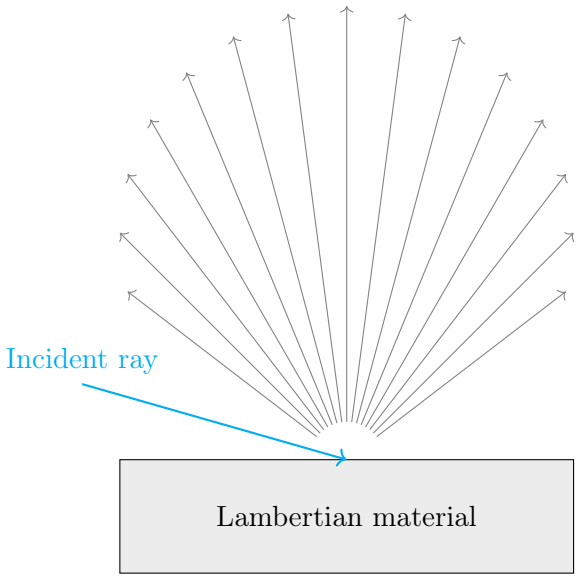


Figure 29: Lambertian material with reflection distribution.

5: The support of *dielectric* materials require the most work. These are transparent, or semi-transparent, materials that both reflect and refract. In real life, each incident ray gives rise to *two distinct rays*; one that is reflected, and one that is refracted. The typical solution to this is to randomly choose to either reflect, *or* refract, with a 50/50 chance for either[2]. This necessitates spawning several rays per pixel in our viewport and averaging the contributions, lest one would experience visual artifacts.

Furthermore, refractions involve trigonometric calculations, due to Snell's law[5]. This has the potential to make the rendering pipeline much more complicated, as these calculations are bound to require creative approximations.

7.2 Microcontroller

The microcontroller currently sends data to the FPGA as soon as its worldview has updated. With more extensive FPGA use, this has the potential to bog down the rendering pipeline. To support more time consuming renders, the microcontroller would have to wait for the FPGA to signal that it is ready for an update.

There are already present interrupt lines between the microcontroller and the FPGA on the TripSitter board, making this a fairly easy software-only change.

7.3 FPGA

Concerning the render pipeline itself, the FPGA is the heavy lifter. This necessarily implies that more changes are required to accommodate the suggested future work described thus far. Most critical are multiple passes of the pipeline by each ray tracing worker. Logic for storing

closest spheres, color contributions so far, and partial frames are therefore the bottleneck as we perceive it.

Integrating more on board memory will be a good first step in achieving the goals described in this section.

To support dielectric materials, trigonometric calculations will be necessary. There are many approximation algorithms available for this purpose, for example the ones defined in `Numerical Recipes`[4]. Benchmarking of these would be required to find suitable candidates.

References

[1] S. Marschner and P. Shirley, *Fundamentals of Computer Graphics, Fourth Edition*, 4th ed., revised. A K Peters, Limited, Taylor & Francis Group, 2016, ISBN: 9781482229417.

[2] P. Shirley, T. David Black, and S. Hollasch. "Ray tracing in one weekend." (Aug. 6, 2023), [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html (visited on Nov. 7, 2023).

[3] Z. Zhongcheng and H. Jingchun, "A novel square root algorithm and its fpga simulation," *Journal of Physics: Conference Series*, vol. 1314, 2019. DOI: 012008. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/1314/1/012008.

[4] e. a. William H. Press, *Numerical recipes: the art of scientific computing*, 3rd ed. Cambridge University Press, 2007, ISBN: 9780511335556; 9780521880688.

[5] H. D. Freedman Roger A.; Young, *Sears and Zemansky's university physics; with modern physics, 12th ed.* 12th ed. Addison Wesley, 2007, ISBN: 9780321501219; 9780805321876.