# Digital ASIC Fabrication

Design Document for Team sdmay26-24

Colin McGann - Project Lead
Samuel Forde - PCB & Layout Lead
Michael Drobot - Firmware Lead
Jack Tonn - Testbench and Validation Lead
Dawud Benedict - Toolflow Lead
Emil Kosic - Repository and Coding Standards Lead
Joshua Arceo - Client/Advisor Communications Lead

Team Email: sdmay26-24@iastate.edu
Team Website: sdmay26-24.sd.ece.iastate.edu

Revised: 9/30/25 – Version 0.1

# Executive Summary

# Learning Summary

# Contents

# List of Figures

# List of Tables

# Definitions

**Bringup** Testing and developing software for a finished and taped out design.

**Caravel** The platform provided by ChipFoundry that contains our design. Includes a RISC-V core, IO protections, memory for the RISC-V core, a Wishbone IO bus, UART, and SPI.

**Connectivity** How the vertices of a model are connected together to form triangles.

**Fragment** A single pixel outputted by the rasterizer.

**Framebuffer** Region in memory that holds a frame of video. The frame could be in progress or complete.

**GPU** Graphics Processing Unit. Dedicated hardware to accelerate graphics calculations.

**GPU Memory** The external QSPI memory used for model, texture, and framebuffer storage.

**Hardening** Compiling a hardware design into fabrication files containing the silicon traces on the die.

**Index Buffer** List of indices in the vertex buffer that make a triangle on a model. Stores the connectivity of a model.

**ISA** Instruction Set Architecture

**Management Core** The RISC-V core included in the Caravel harness, used to initialize and control the shader cores.

**PKBus** Custom multi-master multi-slave bus used exclusively in the user area. Includes an arbiter.

**PMOD** Add-on modules for Digilent FPGAs like the Arty A7 and Zedboard. Connects over 2x8 0.1" headers with a standard spacing.

**Rasterization** The process of converting continuous lines into discrete pixels on a screen.

**RTL** Register Transfer Layer, a common method to simulate hardware modules before synthesis.

**Shader** A small program that runs on a GPU.

**Shader Cores** The custom programmable cores used to run shader programs.

**Synthesis** Compiling the hardware design into logic gates and flip-flops and optimizing out unused gates. Part of the hardening process.

**Tapeout** Fabricating the final design.

**Texel** A texture pixel. One pixel in the texture image.

**Texture** An image that is applied to a model to color it. Usually there is one texture image per model.

**Texture Coordinates** The location in the texture image that corresponds to each vertex on the model. In other words, the mapping between the model and the texture.

**User Area** The part of the finished die allocated to our custom design. Has 10mm$^2$ of area. Everything else on the Caravel harness is provided to us and is not modifiable.

**Vertex** A point in 3D space.

**Vertex Buffer** List of vertices that make up a model. May also hold other metadata for each vertex, such as texture coordinates.

**Wishbone Bus** The data bus included in the Caravel harness. Connects the user area to the management core and allows the user design to be memory-mapped to the management core.

# 1 Introduction

## 1.1 Problem Statement

Many modern graphics processing units (GPUs) are heavyweight high-power products; They take up a considerable amount of space in a computational network and have a high cost. For our project, we have elected to design and test a small footprint educational GPU through the Iowa State ChipForge organization's toolflow. This organization's focus is to give students an opportunity to experience application-specific integrated circuit (ASIC) design, and the toolflow is an open-source solution to design ASICs.

Our GPU is an educational GPU in the sense that it is not the fastest or most powerful GPU solution, but it will be used to help students explore GPU design in a more consumable way than self-research and exploration. Our documentation and weekly reports can be used by students to help them understand the architecture choices we made and to help them consume the design in modularized pieces. Additionally, at Iowa State University, there is a lack of formal instruction on hardware design for graphics. CPRE 4800 "Graphics Processing and Architecture" is the only computer architecture course that gives context on GPU design, but it is seldom offered. Students at other universities may not even have a course on GPU design offered and could use this document to help themselves learn key concepts for a relatively simple GPU design.

## 1.2 Intended Users

Following is a list of potential users for a small-scale GPU.

### 1.2.1 Embedded GPU Users

As our GPU has less logic and a smaller device count, it will take less power to drive it. Thus, applications where power has to be limited, such as an embedded system, could benefit from our design. Embedded GPU Users may not necessarily be searching for the fastest or highest resolution GPU, but instead something that is low power and easy to integrate into a system. As our GPU fits on a 10mm2 die, it has a very small footprint and can fit on most embedded systems with even a small amount of available space.

### 1.2.2 Chipforge Members

Members of ChipForge are students who have shown an interest in the ASIC design process. By being presented with a completed, programmable GPU design, members can dissect the choices of our group, suggest optimizations, and implement those optimizations themselves. By being presented with an educational GPU design, members can add their own ideas and inspirations to the design. For example, a student could write a program that creates an interactive GUI or renders text on the screen, using this GPU as their hardware. Students could also create their own shaders to modify the image output.

### 1.2.3 ISU Faculty

ISU Faculty educate a wide range of topics, and having an open-source small GPU invigorates students to create their own designs. Similar to the i281 CPU used in CPRE 2810 “Digital Logic” to motivate students to learn digital logic, our GPU can be used to motivate students in CPRE 4800 “Graphics Processing and Architecture”, CPRE 4880 “Embedded Systems Design”, or CPRE 381 “Computer Organization and Design” to show design choices and dataflow outside of the course content.

# 2 Requirements, Constraints, and Standards

## 2.1 Requirements & Constraints

### Functional Requirements

- Must be able to render a 3D model with textures to a screen at a minimum frame rate of 15Hz
- Must support 320 x 240 resolution
- Must be able to output to a monitor over VGA

### Technical Requirements

- Written in Verilog HDL *(constraint)*
- Maximum core clock frequency of 40 MHz *(constraint)*
- Design should pass LVS & DRC before fabrication *(constraint)*
- Must use the 130nm Skywater process *(constraint)*
- Minimum 8-bit color depth
- Minimum texture size of 16 x 16 pixels
- Must be able to render at least 4096 triangles per scene

### User Experiential Requirements

- GPU cores must be programmable by the user
- Must be able to take commands from an outside source
- Must have a configurable output resolution
- Must have provided driver code written in C, including a full example of loading and displaying a 3D model with textures
- Must feature debug registers in critical sections of the graphics pipeline accessible from the management core
- Must provide a guide for working with the GPU

### Physical Requirements

- Must fit in a 3mm x 3.6mm user project area *(constraint)*
- Must not use more than 38 GPIO pins *(constraint)*
- Must function nominally at room temperature
- Must not exceed maximum GPIO frequency of 50MHz *(constraint)*

## 2.2 Engineering Standards

Engineering standards ensure that across the industry, engineers agree on processes and procedures when designing products. This allows other professionals to be able to use their products better and increases the user experience by making similar products behave similarly. This allows engineers to have consistency in their work from one product to another and allows companies to build off of these standards instead of having each company recreate the standards for themselves. The following is a list of some of the IEEE standards that apply to our project.

### 2.2.1 IEEE 1364.1-2002: IEEE Standard for Verilog Register Transfer Level Synthesis

This standard describes how to write Verilog code that will be synthesized in a physical design. Our project is written in Verilog, and our goal is to receive a die of the design. Thus, our design must be able to synthesize and harden into a physically manufacturable circuit.

### 2.2.2 IEEE 1364-2001: IEEE Standard Verilog Hardware Description Language

As per the Chipfoundry flow, our design will be written in Verilog HDL. This standard defines Verilog HDL, including verification, timing, and synthesis. Thus, we must follow the standard to be able to write Verilog code that will compile, simulate, and eventually reach synthesis and layout.

### 2.2.3 IEEE 1149.7-2009: IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture

This standard describes test logic for integrated circuits for multiple forms of testing, and it provides a common way to test interconnections between integrated circuits that have been placed on a printed circuit board. This is related to our project because we will have an integrated circuit that will need to be tested and will need to be able to work with other ICs on a PCB, such as memory modules and processing devices.

# 3 Project Plan

## 3.1 Project Management/Tracking Procedures

For our project, we have elected to use the waterfall project management style. This is because we have certain tapeout deadlines that must be met, such as the November and April tapeouts. Due to the nature of the chipfoundry delivery dates, we will not know if our modules are silicon-proven until after the next tapeout. Thus, we feel that the common goals of agile are not as relevant to our project, and we need to have our design mostly to fully realized before we start creating the submodules.

We track our progress by keeping a list of tasks to do and tasks completed. This list is stored in a OneNote and is updated during our weekly meetings. Additionally, our project is stored in a Git repository through GitLab, so we create branches for individual modules and use merge requests to ensure quality HDL and code is submitted.

## 3.2 Task Decomposition

1. Install toolchain and do ChipForge tutorials
   (a) Complete the blinky, UART, and Wishbone Adder tutorials
   (b) Edit the Wishbone Adder to get used to the submodule organization
   (c) Create a custom user project to understand module design from scratch
2. Define project specifications
   (a) Decide what digital ASIC we will implement
   (b) Decide and setup a specific verification method
   (c) Declare what the inputs and outputs of the software are
   (d) Declare what the inputs and outputs of the hardware are
3. Define project design
   (a) Create block diagram to describe flow of data
   (b) Define what operations happen in software and what happens in hardware
   (c) Write ISA to run on the specified cores
   (d) Decide on what peripheral modules may be needed
4. Develop hardware to run $\mu$GPU[1]
   (a) Create PMOD PCB design for external memory
   (b) Test design with FPGA SPI controller and VGA controller

5. Develop software to run $\mu$GPU[1]

   (a) Learn how to parse through Wavefront object file (.obj)<br>
   (b) Define where data will be stored in memory<br>
   (c) Store textures, indices, and vertices in memory

6. Write and test Verilog modules[1]

   (a) Assign each module to a member, and the tests for that module to another member<br>
   (b) Write the Verilog module & SVUnit tests in parallel<br>
   (c) Ensure module passes quality tests<br>
   (d) Assign a 3rd and 4th person to review the module before it is merged

7. Connect and test Verilog modules in the top level design[1]

   (a) Once a submodule is verified, implement within a larger module<br>
   (b) Verify the larger module with SVUnit<br>
   (c) Ensure top level design and software work as expected

8. Test synthesis options

   (a) Ensure designs pass timing<br>
   (b) Calculate maximum possible clock speed of GPU<br>
   (c) Decide what PnR (Place and Route) technique should be used

9. Complete pre-check and layout

   (a) Ensure the entire GPU design fits within die area<br>
   (b) Ensure design passes DRC (Design Rule Check) and LVS (Layout Versus Schematic)

10. Submit for tapeout

    (a) Ensure design is valid and passes Skywater DRC<br>
    (b) Submit to chipfoundry

11. Write user guide

    (a) Write procedures for validation and bring-up once the ASIC is shipped<br>
    (b) Write a guide on how to use the $\mu$GPU

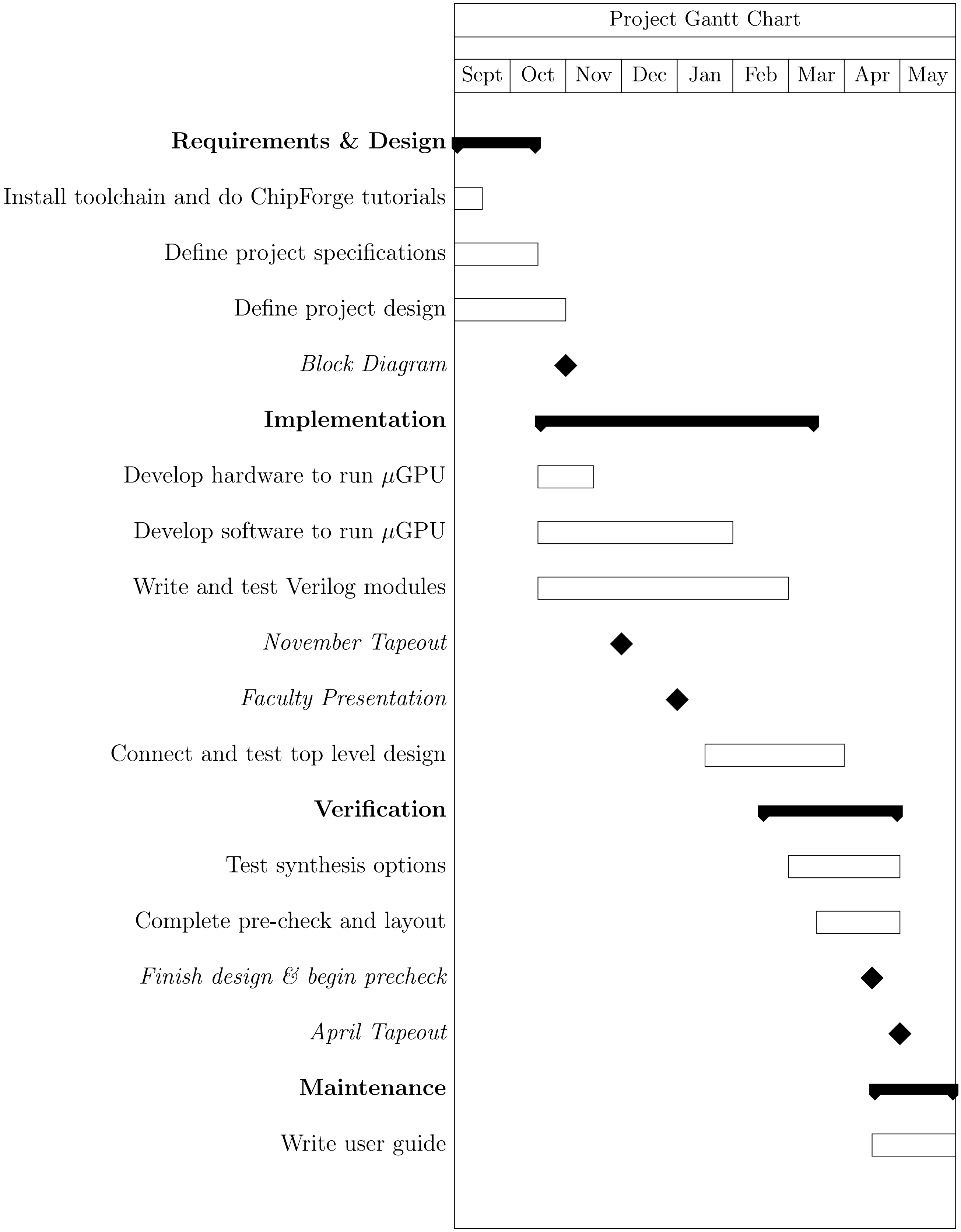
[1]: Tasks done in parallel

## 3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

- Block Diagram
  - Complete GPU block diagram with all required modules
  - Get Client and Advisor approval of design
- November Tapeout
  - Finish verification of rasterizer and bus components
  - Add all verified designs in layout and pass DRC/LVS
- Faculty Presentation
  - Finalize GPU design for presentation
- Finish Design and Verification. Begin Final Precheck
  - Get entire GPU optimized to fit within die area
- April Tapeout
  - Complete precheck, layout, and pass DRC/LVS
- Industry Review Panel
  - Finalize Validation plan
  - Complete software implementation for a visual presentation

## 3.4 Project Timeline/Schedule

The following is a Gantt chart of the tasks listed above, with the bolded categories being waterfall groups, and diamonds being key milestones


Project Gantt Chart
Sept
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Requirements & Design
Install toolchain and do ChipForge tutorials
Define project specifications
Define project design
Block Diagram
Implementation
Develop hardware to run $\mu$GPU
Develop software to run $\mu$GPU
Write and test Verilog modules
November Tapeout
Faculty Presentation
Connect and test top level design
Verification
Test synthesis options
Complete pre-check and layout
Finish design & begin precheck
April Tapeout
Maintenance
Write user guide

## 3.5 Risks and Risk Management/Mitigation

Most of our major risk comes from our fabrication requirements. This includes fitting the design within the die area, meeting our functional requirements with the permitted frequency, and submitting by the tapeout deadline. Another major risk is possible errors during fabrication. There is a non-zero chance that the chip that is returned has issues that cannot be found unless exhaustive validation is done. In this case, this risk is completely out of our control, but having a validation plan can prevent further issues.

In addition to fabrication, the risks fall mainly on design and implementation. Most of these risks are simply the case that the module is not completed or a failed verification. These risks would have great consequences on the project, since every component needs to be present for the GPU to work as expected. Although the big impact has been observed, the probability that a design is not finished is much lower than other risks due to DRC, LVS, or area constraints.

Although our end goal is fabrication of the $\mu$GPU, the impact of fabrication risks do not make this project meaningless. The design itself will still benefit Chipforge members, allowing them to learn and build off GPU architecture. In this case, most risks are that a functional component does not work as intended. We plan on reducing these risks significantly by making sure each module is verified by a different member through SVUnit.

| **Risk** | **Probability** | **Project Impact** |
|---|---|---|
| Chipfoundry closes or doesn't allow tapeout | 5% | High |
| Chipfoundry deadline not met | 10% | High |
| Cannot optimize design to fit the die area | 20% | High |
| Cannot optimize design for 15Hz framerate | 30% | Medium |

Table 1: Project Risks

## 3.6 Personnel Effort Requirements

These are the predictions of our time requirements for the $\mu$GPU, which is based on our initial weeks working on senior design and past ChipForge experience. This is also based on the work of previous senior design teams with projects in ASIC design.

| **Task** | **Time Estimate (hours)** |
|---|---|
| Install toolchain and do ChipForge tutorials | 35 |
| Define project specifications | 15 |
| Define project design | 40 |
| Develop hardware to run $\mu$GPU | 20 |
| Develop software to run $\mu$GPU | 50 |
| Write and test Verilog modules | 100 |
| Connect and test Verilog modules in the top level design | 25 |
| Test synthesis options | 80 |
| Complete pre-check and layout | 40 |
| Submit for tapeout | 10 |
| Write user guide | 90 |

Table 2: Personnel Effort Requirements

## 3.7 Other Resource Requirements

Our extra resource requirements will consist mostly of the equipment that can be found in the senior design lab and Chipforge lab. This includes oscilloscopes, logic analyzers, FPGAs, and computers. All of these resources are freely available to use, so will not be a problem to attain.

# 4 Design

## 4.1 Design Context

### 4.1.1 Broader Context

Modern GPUs are large, expensive, and power hungry. We will be providing a small form factor, energy efficient alternative to these GPUs, called a $\mu$GPU for our user groups. Modern GPUs are attractive to many problems due to their ability to run parallel code in their cores. We have made our pipeline programmable to allow for users to map their own problems to the $\mu$GPU.

Our design is to support development of ASICs and to teach about graphics pipelines to ChipForge members. Additionally, we have made the pipeline programmable so it can be mapped into courses from ISU faculty. Finally, embedded GPU users may find enjoyment in our design due to its small size and light power consumption.

Our project addresses the need for open source ASICs. ChipForge members enjoy open source ASICs because they allow members to start off with a functional code base, which allows members to see what Verilog synthesizes and what may not, and how to solve common problems. This is introducing the members to the larger open source digital design community, which will also benefit from our open source GPU design.

Additionally, some special considerations are explored in Table 3.

| Area | Description | Examples |
|---|---|---|
| Public Health, Safety, and Welfare | Our project provides a small GPU that does not currently exist to most of our user groups, namely ISU Faculty and ChipForge members. Through Chipfoundry, we are lowering the barrier of access to a small scale GPU. | The creation and distribution of a product like the $\mu$GPU increases the job market in terms of design and testing, and marketing. |
| Global, Cultural, and Social | Our project helps grow and support the open source digital design community, thus creating a more active digital design community. This community is an implied user group through Chipforge members and Embedded GPU enthusiasts. | By adding to the digital design community, we are growing the group and introducing ChipForge members to a larger community. |
| Environmental | Our project requires silicon wafers to be manufactured, which must be mined out from the Earth and delivered to the foundry, which has a net negative environmental impact. Additionally, the chips must be shipped across the United States, increasing the environmental cost of our product. | Because the $\mu$GPU is smaller than modern GPUs, it will have less components, thus less waste in manufacturing and when it reaches end of life. It being lighter also reduces the environmental cost of shipping, thus lowering the environmental impact. |
| Economic | The $\mu$GPU is an open source design that any group could copy and modify to fit their needs. This makes the cost of development of future revisions less. | By creating and fabricating this open source design, we are supporting the growth of small scale ASIC development. With this growth, individuals can create and market their own products, thus increasing market competition, which lowers price. |

Table 3: Broader Context

### 4.1.2 Prior Work/ Solutions

This project gave us the autonomy of choosing the design we wanted. When thinking of considerations, we tried to think of what would benefit Chipforge the most. Many Chipforge members have an interest in GPUs, and with the class CPRE 480: Graphics Processing and Architecture no longer being offered, a $\mu$GPU would provide members a great starting point for learning about GPU hardware design.

Our starting consideration was the GPU that is implemented when taking CPRE 480. The lab portion of this class involves designing a fixed graphics pipeline with rasterizer. However, with a larger team and longer timeline, this was evolved to become a multicore programmable pipeline, with vertex and fragment shading.

Rasterization is the process of turning object vectors into pixel images. Although more complex rendering methods exist today, rasterization is still commonly used due to being fast. This means the rasterizer is one of the most important modules in the $\mu$GPU, and it was the first module to be created. Its design was modeled after older NVIDIA rasterization techniques, which use barycentric coordinates to determine if a point is inside a triangle.

### 4.1.3 Technical Complexity

Our design contains several components that have non-trivial amounts of complexity:

- Programmable Cores
  - We need to implement what is essentially an entire processor that will handle performing any of the number of operations passed to it. We will also need to route the data in and out of it in a sensible way.
- Rasterizer
  - The rasterizer will handle converting a set of 3D coordinates to an array of 2D pixels. This consists of a set of math operations that will need to be executed efficiently for performance reasons. It also handles mapping textures to the polygons.
- VGA Controller
  - We decided to write our own VGA controller for better customization and integration into our overall design. Building the controller out will require properly timing output signals to display pixels within the standards established for VGA displays.

## 4.2 Design Exploration

### 4.2.1 Design Decisions

There are several important decisions that went into planning out our design:

1. Pipeline Flexibility
   - For the design of our graphics pipeline, we had to decide between a fixed pipeline or a programmable one.

2. Design Verification Framework

   - The ChipForge design flow we worked with doesn't formally have any specific verification standards. As a part of a push for better verification standards both in our senior design group and in the ChipForge extracurricular, we decided to implement a specific verification framework to use with all of our Verilog designs.

3. Instruction Set Architecture

   - An important part of creating a programmable graphics processor is the set of instructions that the graphics processing cores will support. This includes any operations necessary support programmable operations that we are looking to make available.

4. Programmable Core Count

   - Since we decided to make our pipeline programmable, that means that we have cores that support a range of operations. The next step in that process was deciding the number of cores.

### 4.2.2 Ideation

1. Were we to choose to build a fixed pipeline, the complexity of the pipeline would be dramatically simpler, at the cost of customization. The programmable pipeline would have more flexibility but would be much more complex to design and would be more difficult to program for as well.

2. For verification the main options we looked at and their considerations were:

   - **CocoTB:** Already implemented in ChipForge's Caravel harness, and relatively easy to use.
   - **SVUnit:** Lightweight and easy to implement in existing toolflow. Flexible with what simulator we use
   - **UVM:** Industry standard for HDL verification. Lots of overhead for a single test, but very modular.

3. For designing our ISA, we had to look at what we needed for the various stages we would like to implement. This included operations needed to meet requirements (such as vertex shading) and operations for other goals (such as ray tracing and machine learning).

4. For deciding the number of cores the main thing we looked at was the estimated size of each core. Since we have a small die area for our $\mu$GPU, we decided we would fill whatever space we had left from the rest of the design with programmable cores.

### 4.2.3 Decision-Making and Trade-Off

1. We decided to proceed with designing a programmable pipeline, as programmable pipelines are more modern and better fit our users' needs. Additionally, since we have seven group members, we figured we would have a large enough labor pool to implement the programmable pipeline.

2. For the purposes of this project, we decided to integrate SVUnit into our project, as we liked it's ease of use, and the simulator flexibility won us over.

3. We have yet to fully pick out our ISA, and are actively working on it.

4. We decided we want around six cores in our project, as we think this will give us a level of performance that will meet our expectations.

## 4.3 Proposed Design

### 4.3.1 Overview

Our design is a programmable GPU primarily designed for 3D rasterization. It features:

- Six (6) custom programmable cores following a custom ISA
  - 16kB internal program memory
  - 16 32-bit registers per core
  - 48 32-bit registers shared between all cores
  - Multithreading capability
- One (1) pipelined rasterizer module
- Three (3) external QSPI memory ports
- One (1) external VGA port
- One (1) external control interface
- One (1) RISC-V management core for initializing and controlling the $\mu$GPU

Because the design is primarily optimized for 3D rasterization it includes dedicated rasterization hardware. However it is also capable of other GPU compute tasks such as 2D rendering, ray tracing and machine learning because the custom cores are programmable. The user can write small programs in assembly that run in parallel on the cores to do virtually anything.

#### 4.3.2 Detailed Design and Visuals

The flow of data in the $\mu$GPU is shown in Figure 1. 3D models are input into the system through firmware in the management core or the external control interface and sent to the shader cores model-by-model for each frame. The shader cores and rasterizer render the frame and store it to the framebuffer. Finally, the VGA module takes the finished frame and outputs it to the display.

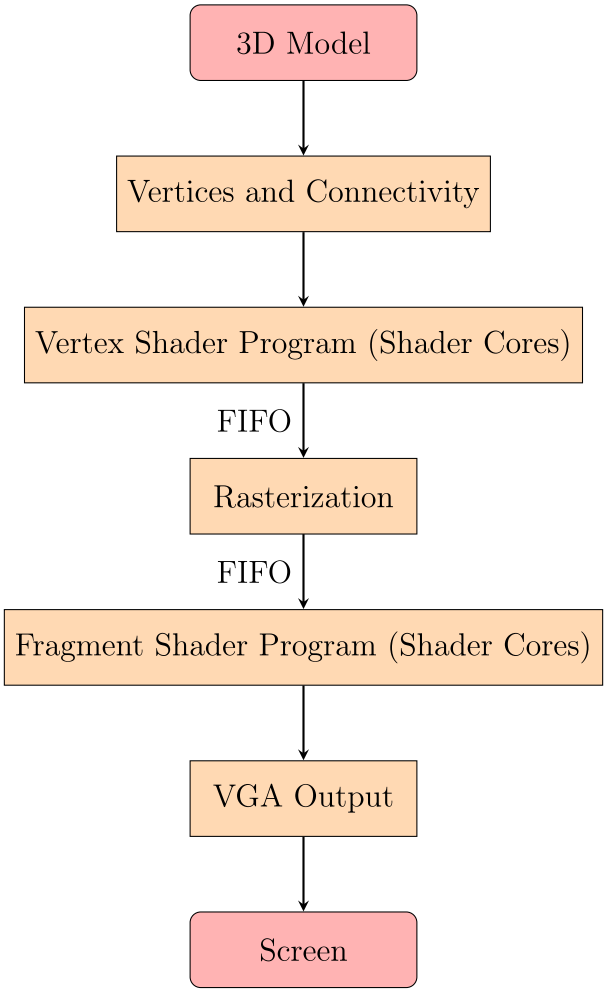

3D Model
Vertices and Connectivity
Vertex Shader Program (Shader Cores)
FIFO
Rasterization
FIFO
Fragment Shader Program (Shader Cores)
VGA Output
Screen

Figure 1: $\mu$GPU Data Flow

##### 4.3.2.1 Input Processing

3D models come in various formats, but they are all essentially a list of vertices, connectivity, and texture coordinates. The management core doesn't have enough power to process these large files. Instead, the host system that implements the $\mu$GPU handles preprocessing the model and converting the format.

The 3D model is stored in GPU memory and organized in the vertex buffer and index buffer.

The formats of the vertex and index buffers are shown in Figure 2. Vertex buffer entries are flexible since loading a vertex is done by the shader cores and is programmable. Index buffer entries are fixed since loading is handled in hardware.

```
typedef fixed_s32_t int32_t; // Custom fixed-point type defined in hardware

// Sample vertex buffer entry without texture coordinates
typedef struct __packed {
    fixed_s32_t x;
    fixed_s32_t y;
    fixed_s32_t z;
} vertex_t;
vertex_t vertex_buffer[N_VERTICES];

// Sample vertex buffer entry with texture coordinates
typedef struct __packed {
    fixed_s32_t x;
    fixed_s32_t y;
    fixed_s32_t z;
    fixed_s32_t tx;
    fixed_s32_t ty;
} vertex_t;
vertex_t vertex_buffer[N_VERTICES];

// Index buffer entry
typedef struct __packed {
    uint32_t idx[3];
} index_t;
index_t index_buffer[N_TRIANGLES];
```

Figure 2: Vertex and Index Buffer Entries for 3D Models

#### 4.3.2.2 Shader Cores

The shader cores are custom-designed pipelined cores that implement a custom ISA. They can be programmed in assembly to do anything the user wants, enabling functionality like machine learning, ray tracing, GPGPU compute, and physics simulations. The custom ISA allows the cores to implement complex functionality and pack the functionality into a single fast instruction. It currently supports vector operations (dot product, add, subtract, scale), vector loads, and pushing and pulling from the rasterizer in addition to standard scalar operations (add, subtract, add immediate, etc).

Each core has 16 word-length registers in a private register file. These hold information relevant to the data that the core is currently working on: vector coordinates, colors, normal vectors, etc. All cores also have access to 48 word-length global registers which hold common data for each model, such as the model-world-projection matrix. The register usage is defined by the user through the shader program.

The instruction memory (IMEM) and program counter (PC) is shared between all cores, meaning all cores are executing the same instruction at the same time. The complexity of individual PCs and instruction fetch logic is too much for the size of the shader cores. However, users still want to have branches and jumps in their programs. Even in basic rasterization the cores need to be split between vertex and fragment shading depending on whether there are vertices remaining in the model or fragments ready from the rasterizer.

Multithreading is handled using predication, where instead of modifying the PC for a jump or branch the execution of each instruction is dependent on a condition encoded into the instruction. In other words, each instruction has `if(predicate) { instruction }` built into the encoding. This keeps the fetch logic simple while allowing conditions. This gets more complex when introducing nested conditionals. This is addressed with a max conditional depth constant.

The following sections cover the common use cases for the shader cores in a rasterization workload. The cores are programmable and are not limited to only vertex and fragment shading, but they are most optimized for these tasks.

#### 4.3.2.2.1 Vertex Shading

Vertex shading consists of 3 steps, which are combined into a single matrix multiplication done in hardware using the shader cores. This is run on every vertex in every model. Vertex shading uses homogeneous coordinates, which add a fourth entry into each vector. The fourth entry is the perspective parameter, usually called $w$.

1. **Model matrix:** Move, scale, stretch, and rotate the vertex from the model coordinate space (determined by the model designer, likely with the origin centered on the model) into the world space containing our scene.

$$v_{world} = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

2. **View matrix:** Translate the vertex into camera space. Essentially a translation matrix but with the camera coordinates specified explicitly.

$$v_{camera} = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} v_{world}$$

3. **Perspective matrix:** Project each vertex into the screen. Gives the screen-space coordinates (in pixels) of each vertex with X and Y and a depth away from the camera with Z. Essentially transforms the vertex into a coordinate system with the origin centered on the camera and positive Z pointing out of the lens.

$$v_{screen} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} v_{world}$$

$$\text{Convert to a 3-vector: } v_{screen} = \begin{bmatrix} x/z \\ y/z \\ z/z \end{bmatrix}$$

All of these matrices can be combined via matrix multiplication into a single 4x4 matrix. This matrix is called the Model-View-Perspective matrix and is provided to the shader cores by the management core. It is stored in the global registers.

The vertex shading process involves loading a vertex from the vertex buffer and applying the Model-View-Perspective matrix. After a vertex has been shaded it is pushed to the rasterizer through a FIFO using a special assembly instruction.

#### 4.3.2.2.2 Fragment Shading

Fragment shading applies lighting and post-processing to all fragments coming out of the rasterizer. The exact algorithms are flexible since the shader cores are programmable. The user could implement basic direct illumination or fancier global illumination or ray tracing algorithms. The user could also implement surface smoothing to reduce sharp edges.

The fragment shader receives the screen coordinates of a pixel, the pixel's color, and the unit normal vector of the surface containing the pixel from the rasterizer. For direct illumination, it performs a dot product to check the angle of the light relative to the normal vector. If the angle is greater than 90 degrees, the surface is facing towards the light and should be colored (the normal vector of the surface and of the light are pointing at each other). Otherwise the surface is in the shade and should be black.

$$\theta = \cos^{-1}(\frac{A \cdot B}{|A||B|}) = \cos^{-1}(A \cdot B) \text{ if A, B are unit vectors.}$$

$$\text{Pixel} = \begin{cases} \text{Color} & \text{if } A \cdot B < 0 \\ 0 & \text{otherwise} \end{cases}$$

#### 4.3.2.3 Rasterization

The rasterizer takes in a stream of vertices from the vertex shader in sets of three. These vertices are then translated into a position on the screen. The rasterizer will determine whether the face is facing toward or away from the camera in a process called back-face culling. It will do this by determining if the vertices provided are in a clockwise or counterclockwise order. This is will be easy to implement in hardware and will allow a large portion of faces sent to the rasterizer to short-circuit.

If this stage passes the upper-left and lower-right coordinates of a bounding box will be calculated for the set of vertices (Figure 3). This bounding box will be the area of pixels that must be iterated to draw the desired triangle.

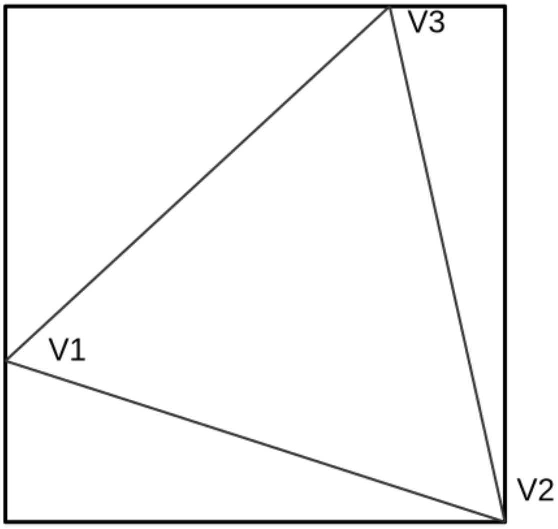


Figure 3: A bounding box around a triplet of vertices

For each pixel in the bounding box the rasterizer will calculate the barycentric coordinates of the pixel within the triangle. The barycentric coordinates are the weights for a weighted average of the three coordinates of the triangle and can represent any point in the triangle. The choice to use this coordinate system works in our favor at multiple points down the pipeline and is somewhat simplistic to calculate the equations below. These coordinates are easy to use to make a decision on whether the pixel is within the triangle or not; the rasterizer can simply check to make sure all coordinates are greater than zero. If the point is not in the triangle, the rasterizer will continue by evaluating the next pixel. If the pixel is in the triangle, the rasterizer will pass this pixel down through the pipeline and continue to the next pixel.

$$\lambda_0 = \frac{(y_1 - y_2)(x - x_2) + (x_2 - x_1)(y - y_2)}{(y_1 - y_2)(x_0 - x_2) + (x_2 - x_1)(y_0 - y_2)} \tag{1}$$

$$\lambda_1 = \frac{(y_2 - y_0)(x - x_2) + (x_0 - x_2)(y - y_2)}{(y_1 - y_2)(x_0 - x_2) + (x_2 - x_1)(y_0 - y_2)} \tag{2}$$

$$\lambda_2 = 1 - \lambda_0 - \lambda_1 \tag{3}$$

Figure 4: Calculation of the barycentric coordinates for a triangle defined by $(x_0, y_0)$, $(x_1, y_1)$, $(x_2, y_2)$ and a point $(x, y)$.

The next stage of the pipeline is the depth test stage. This stage is where the rasterizer will determine whether the new pixel is in front or behind an already drawn pixel. To do this, the depth of the currently drawn pixel needs to be evaluated. This is done by multiplying each of the barycentric coordinates ($\lambda_0$, $\lambda_1$, $\lambda_2$) into the depth of the 3 vertices ($z_0$, $z_1$, $z_2$) and adding the results (Figure 5). The calculated depth is then compared with a value in a special portion of memory called the depth buffer. If the evaluated value is less than that present in the depth buffer, then the current pixel passes the depth test. This means that the new value will be written back to the depth buffer for future rasterization passes, and the coordinate will be passed to the texture unit for texturing.

$$\text{Depth} = \textstyle\sum_{i=0}^{2}(\lambda_i \cdot z_i)$$

Figure 5: Calculation of the depth of a pixel within the rasterized triangle.

The final stage of the rasterization pipeline is the texture unit. This unit evaluates the texture address of each pixel and retrieves it from a texture buffer. The calculation of the texture pixel address is similar to the calculation of the pixel depth (Figure 6). This address can be used to index into the current texture and resolve the base color of the current pixel. This pixel and corresponding information is then sent out through a FIFO to be processes by the fragment shader.

$$t_x = \textstyle\sum_{i=0}^{2}(\lambda_i \cdot u_i)$$

$$t_y = \textstyle\sum_{i=0}^{2}(\lambda_i \cdot v_i)$$

$$\text{Texture Address} = t_y \cdot \text{Texture Width} + t_x$$

Figure 6: Calculation of the texture address for a pixel within a triangle. $(u_{0-2}, v_{0-2})$ are the texture coordinates defined at the 3 corners of the current triangle.

#### 4.3.2.4 Buses, Memory, and Controls

The shader cores, rasterizer, VGA output, and GPU memory are memory-mapped to the management core for control and debug through the Wishbone bus. Wishbone is an open-source memory bus commonly used on FPGA projects. However, it only supports one master. The shader cores, VGA output, and GPU memory are all bus masters. Therefore we created a custom multi-master multi-slave arbitrated bus called PKBus. It is used exclusively in the user area of the design and is connected to the Wishbone bus through a bridge. Configuration registers for each peripheral are mapped to the Wishbone bus directly, but data access for GPU memory (for example) is done over PKBus.

The $\mu$GPU has external QSPI memory chips capable of running up to 133MHz. Graphics assets are very large, the framebuffer alone at our max resolution of 320x240 is 76.8kB. Putting memory onboard the finished die would waste valuable area and restrict our design's capabilities. External memory allows us to store more complex assets and models.

#### 4.3.2.5 VGA Output

Generating a VGA signal involves outputting the HSYNC and VSYNC signals at the correct timings and outputting color over 3 analog lines (red, green, and blue). The $\mu$GPU outputs 320x240 @ 60Hz using the standard timings and 8-bit color depth. 320x240 is a quarter of 640x480. The VGA output still sends out standard 640x480 timings, but performs line doubling and pixel doubling to lower the resolution. This doesn't affect overall performance since each line is read with a single burst read and cached within the VGA output module. A new line of pixels is only read *every other* line. The VGA module also supports lower resolutions: 160x120 (16th-scale) and 80x60 (64th scale).

The conversion from digital 8-bit color to analog VGA color is done using a custom resistor DAC on the die (Figure 8). This saves 5 IO pins (3 for analog RGB instead of 8 for all 8 bits), which are extremely valuable on this chip. The alternative is an external resistor DAC similar to the one used for testing.

#### 4.3.2.6 Hardware

The $\mu$GPU needs some some supporting hardware in addition to the chip design: the aforementioned external memory, a VGA connector, and headers for connecting to a host system. There are two board designs that accompany the silicon:

- **MemoryVGAPmod:** A quad PMOD board used for FPGA testing. Contains 3 QSPI memory chips, a VGA DAC, and a VGA connector.
- **$\mu$GPU:** The carrier board for the final taped out design. Contains an M.2 slot for the Caravel board we receive from ChipFoundry, 3 QSPI memory chips, a VGA connector,

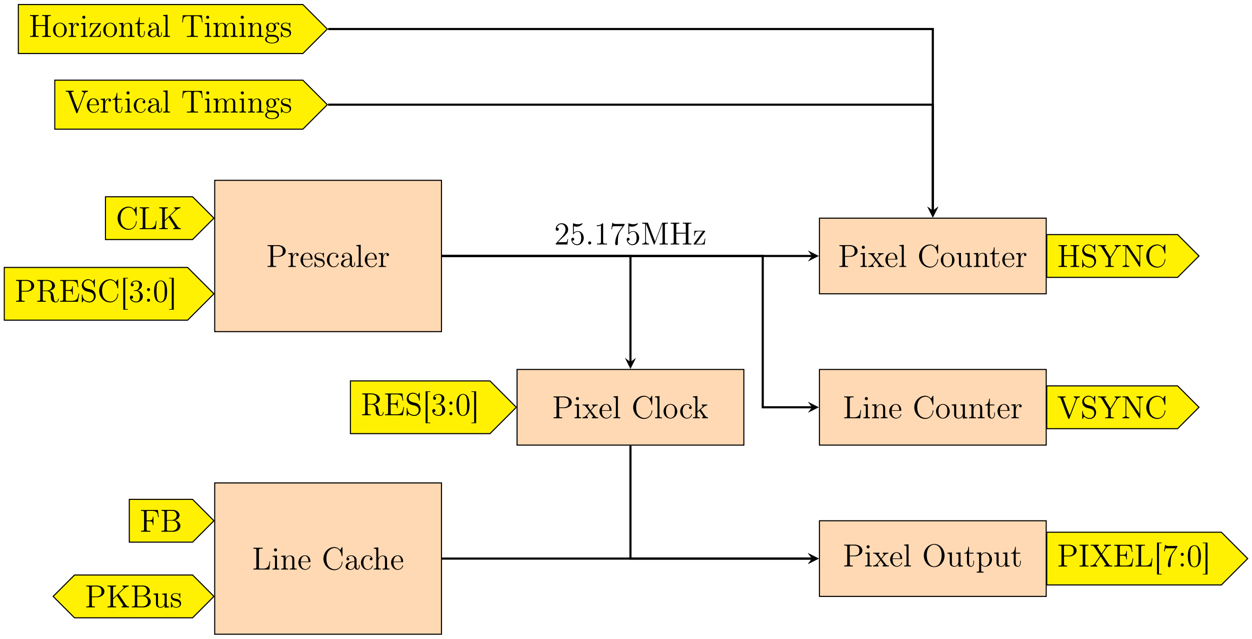


Figure 7: VGA output module block diagram

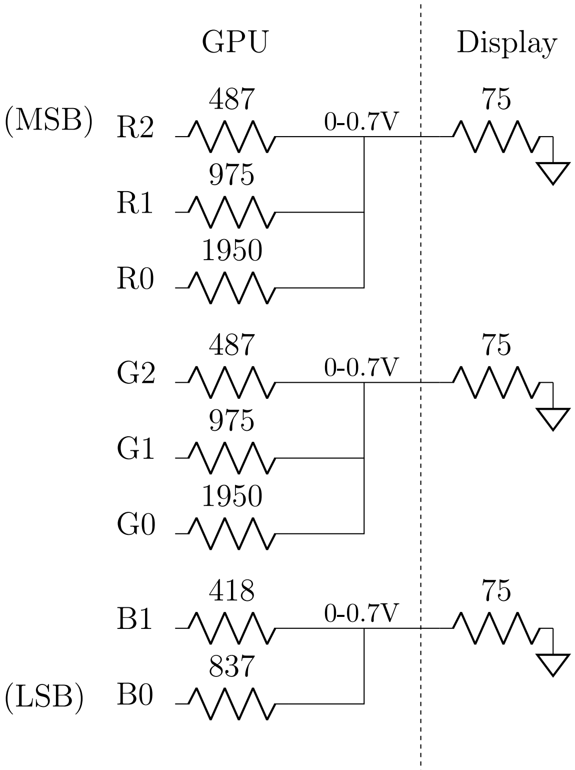


Figure 8: VGA color resistor ladder

a VGA DAC in case the onboard one doesn't work, headers for connecting to a host, and debug headers.

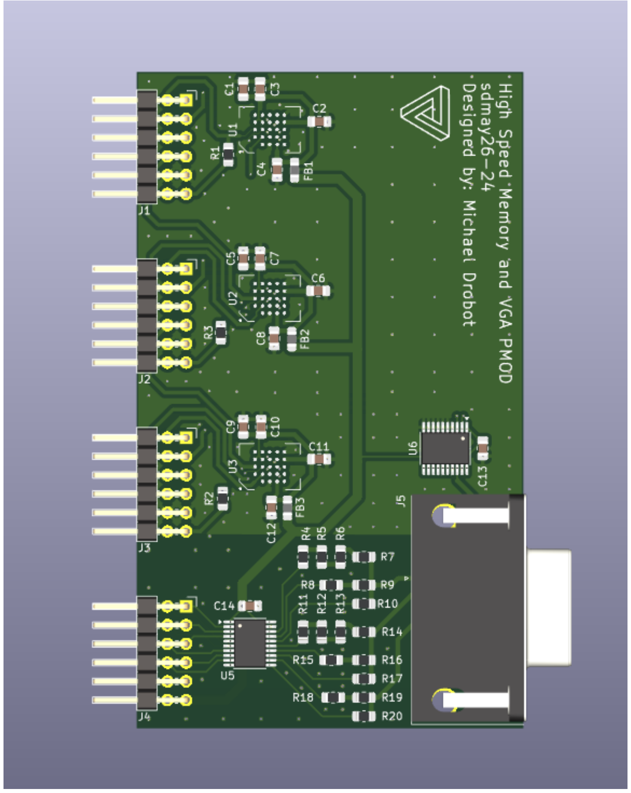


Figure 9: 3D render of MemoryVGAPmod

### 4.3.3 Functionality

In a typical user environment, we would expect that users are using our product as a development or teaching platform. A user would typically use documentation provided by us as a reference to build their own application or obtain software that is already built. The user would be able to build on the software already provided by our team because of the open-source nature of the project.

We provide the user a datasheet that describes the functionality of the chip and register descriptions. Additionally, we provide an ISA summary for the shader cores that details each instruction's functionality and encoding. The build system used for the development process is open-source so bring-up should be relatively straightforward.

### 4.3.4 Areas of Concern and Development

Our biggest concern with regards to technology is the area constraints placed on us by the multi-project wafer service, chipfoundry.io. We have 10 $\text{mm}^2$ of space on the die, which is significantly less area than any commercial graphics co-processors that we have investigated. Additionally, we have found that our process generally has a larger feature size and fewer layers than other graphics accelerators.

To attempt to combat against issues with die use we will design our system in a modular way. This will allow us to scale the processing power of the graphics core with the size of the die. This would, of course, harm performance but we are willing to make sacrifices in performance to provide a more complete and adaptable product to the user.

## 4.4 Technology Considerations

The primary technologies we will be using in our design are the SKY 130 PDK, SVUnit, EFabless Caravel, and OpenROAD. These are all open-source developments which we will use to design and fabricate our GPU. The main concern we have with these technologies is a lack of documentation and a possible lack of quality when compared with some closed-source counterparts.

We will be addressing this by attempting to use some closed-source software for our fabrication such as Cadence. This will allow us to get the most out of our PDK.

## 4.5 Design Analysis

Currently we have been making progress designing the rasterizer, memory interface, VGA controller, and Wishbone configuration system. We hope to integrate our components together soon to prepare for the upcoming November 2025 tapeout.

We believe that we will only have to make minimal modifications to our original design plan. We have been able to design a large part of the product so far and have not had any issues come up that would warrant a full design revision.

# 5 Testing

## 5.1 Unit Testing

## 5.2 Interface Testing

## 5.3 Integration Testing

## 5.4 Regression Testing

## 5.5 Acceptance Testing

## 5.6 User Testing

## 5.7 Results

# 6 Implementation

# 7 Ethics and Professional Responsibility

## 7.1 Areas of Professional Responsibility/Code of Ethics

## 7.2 Four Principles

## 7.3 Virtues

# 8 Closing Material

## 8.1 Conclusion

## 8.2 References

## 8.3 Appendices

# 9 Team

## 9.1 Team Members

## 9.2 Required Skill Sets for your Project

## 9.3 Skill Sets Covered by the Team

## 9.4 Project Management Style Adopted by the Team

## 9.5 Initial Project Management Roles

## 9.6 Team Contract