Spinning LED Display

ECE532-2023S FINAL REPORT

SPENCER BALL | NIKOO GIVECHIAN | KEVIN KIM | EDDIE TIAN

1. Table of Contents

2. Overview

2.1. Goals

This project aims to design and implement a holographic display via a strip of LEDs rotating at high speed to create the illusion of an image. Such designs have an advantage over standard LED panel displays as it permits light through the display, creating the illusion that the image on display is holographic.

Figure 2-1. Production version of the holographic spinning display

Similar designs exist; however, they lack user interactivity with clunky software for transferring images according to reviews. We would like our project to achieve a smooth image, like commercially available products, and incorporate improvements such as manipulating the hologram with user input from peripherals.

2.2. Reference Design

This design was based on what can be seen in the video here. While this production version has very high resolution, the goal with this project was to be able to display any 64 x 64 bit image, as we decided to use LED arms with 32 LEDs per rod (radius).

2.3. Block Diagram

A higher-level overview of the design can be seen in the figure below. The design was split into 4 blocks: image input, image transformer block, LED driver, and the Microblaze. The image input block would be responsible for getting input image data from an SD card. The image transformer block would take convert the image from cartesian to polar, so that it can be sent to the LED driver in the form of RGB values per LED on the LED rod for however many times the LED rod values would be updated each rotation. The LED driver was responsible for taking in the image, and updating the LED values at the correct time. Finally, the Microblaze would involve the motor driver and encoder in order to power and keep track of LED arm angles.

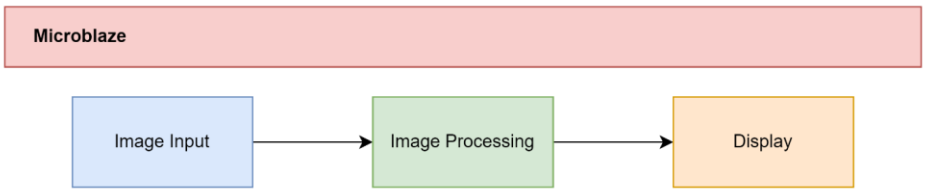

Figure 2-2. Overview of the design

3. Outcome

3.1. Results

This project was integrated successfully with two of the blocks being integrated, and 2 of them not being fully integrated by the end. The image input and image transformer blocks were functional and we did integrate the two and test them in simulation, but we ran out of time to integrate these two blocks with the LED driver and microblaze. In the end, what we presented was a hardcoded image on the microblaze and LED driver being integrated with that.

The image input block can read images from an SD card and place data on an AXI bus, but due to a missing synchronization signal, the other parts are unable to communicate with the first block.

Because an image from the SD card could not be acquired, a flower was hard coded into BRAM memory and displayed on the LED arms.

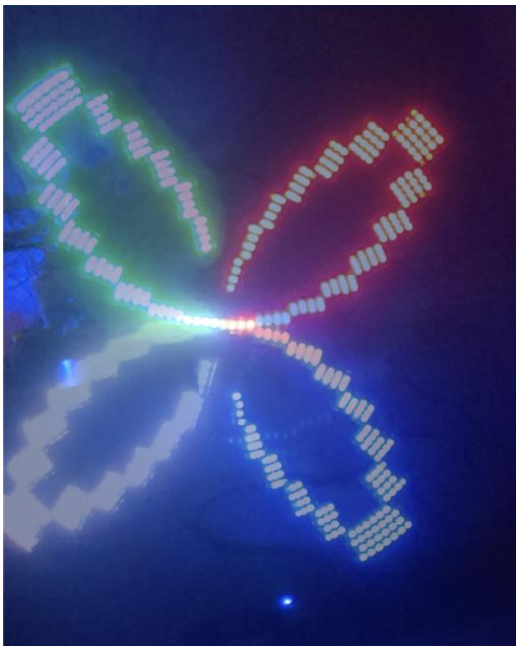

Figure 3-1: Final output of project. Two separate frames from a video were overlaid.

3.2. Further Improvements

Further improvements would include getting the image input block to interface properly with the rest of the design, then adding user input to the image to be able to manipulate the image. The image also has some angular jitter, meaning the displayed image randomly rotates within 20 degrees. The encoder's status register also sometimes has the wrong value set. We believe this is because of electromagnetic radiation from the encoder's magnet influencing the status register. This could be solved by periodically rewriting a value to the status register.

Our design initially had 4 led arms but during the demo we only had 2 arms connected. A future improvement would be to add the remaining two arms, thus allowing us to reduce the motor speed and hopefully reduce the angular jitter as well.

4. Description of Blocks

4.1. Image Input Block

This block is responsible for taking data from an input source, then extracting the bitmap and writing it to an AXI bus.

4.1.1. SD Card Reader

The SD card reader uses the XESS Corp SD card IP to interface with the SD card. This IP provides an interface where an address and read request is provided to the IP, and it retrieves a sector (512 bytes) of data from the SD card in one-byte sections with handshake signals to control data flow.

4.1.2. BMP File Reader

Once one sector of data is read, this custom IP block examines the header to confirm if the data is of the BMP type, the offset of the bitmap from the start of the file, the colour depth of the file, the width and height, and the size of the file. Once the header is read, the bitmap is read, and each pixel's order is rearranged. The protocol used by our LED strips needs colours in GRB order, but colors are stored in RGB order in the bitmap. Each pixel's colour depth is also truncated to 6 bits from the original 24 bits to reduce board resource usage by later blocks as the full 24-bit colour depth would exceed what the FPGA is able to provide. If the end of the sector's address is less than the initial address plus the size of the file, another read is issued to the SD card and the bitmap is processed for that sector as well. This is repeated until the entire picture is read.

4.1.3. AXI Master

Once a pixel is read by the BMP file reader, the 6 bits are padded with 2'b00 then sent to the next block via a custom AXI-lite master. The master writes to a predetermined location in memory which was selected when configuring the block in Vivado and uses a strobe value of 4'b0001 to only write 8 bits of data and save bandwidth.

4.2. Image Transformer Block

The purpose of the image transformation block was to take in the image input from the SD card, and output it to the LED driver in polar coordinates. The design involved 32 24-bit LEDs per arm, and every 32 * 24 bits outputted to the LED driver corresponded to the pixel values for the LEDs at some rotation angle theta. A diagram below summarizes this concept.

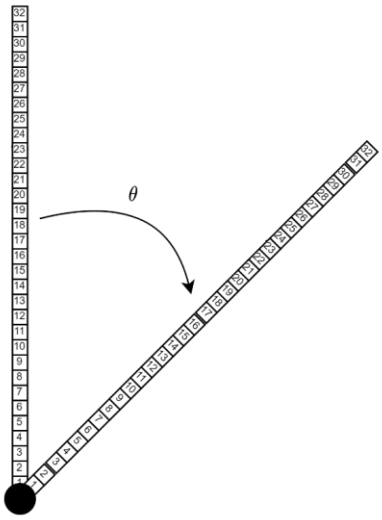

Figure 4-1. LED arm representation. 32 LEDs per arm, theta represents angle between LED arms when the arm is rotated.

A higher-level overview of this block is outlined in the figure below.

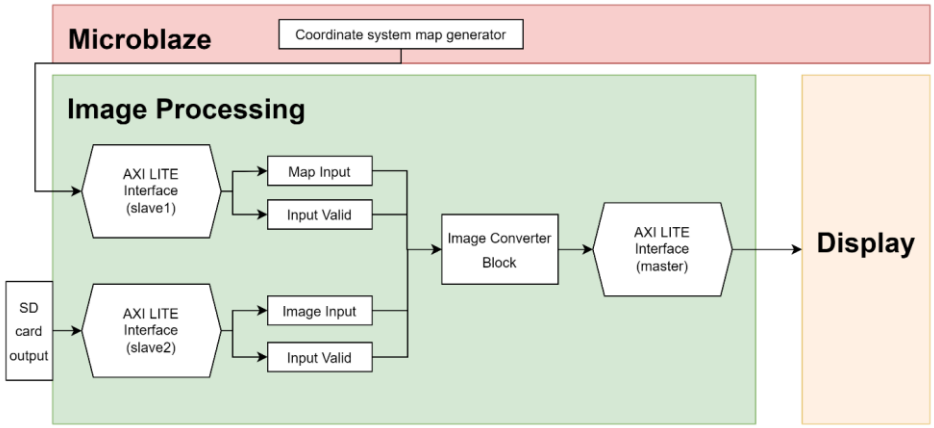

Figure 4-2. Overview of the Image Mapping Block

4.2.1. AXI Slave 1

There are 2 AXI slaves in this block. The first slave takes in a map array from the Microblaze, where the map is generated via C-code and is what will be used to recover the correct LED values for each arm LED in the Image Converter Block. The microblaze additionally sends an input valid signal once it is finished sending data.

Since the image map is significantly larger than the 32 bits that can be sent over AXI, a bitshift is used to write the entire image map to one register (of size 64 * 64 * 16, where 64 is the dimensions of the input image, and 16 bits are used to hold the x and y values which are placed in the map). The slave constantly writes to the last 32 bits of the register, and shifts by 32 bits afterwards. This repeats until the entire image map is received, which is determined by counting the number of bitshifts.

4.2.2. AXI Slave 2

The second slave interface receives the input image from the SD card block. It works in a similar manner with bitshifts and an input valid signal being sent by the SD card block when it is done transmitting.

4.2.3. Image Converter Block

This block is responsible for taking the completed map and input image, and using those to generate the output image to be sent to the LED driver. It works by iterating over the entries of the map, looking up the pixel values of the input image at each x, y retrieved from the map, and placing that pixel value in the output image register. The map was created such that the resulting output would be a rod of LED values for some number of positions per rotation. The angle between each of these positions is what I referred to as θ, and the output image array is of size $\theta * 32 * 24$, where there are 32 LEDs per arm and 24bits per RGB value. Once the output image is completed, an output value signal is set, which indicates readiness to send the output map to the LED driver.

A representation of what is being sent to the LED driver block can be seen below. This was generated via a python script to check the validity of the idea, and was calculated with $\theta = 2°$ (180 updates of LED values per rotation).

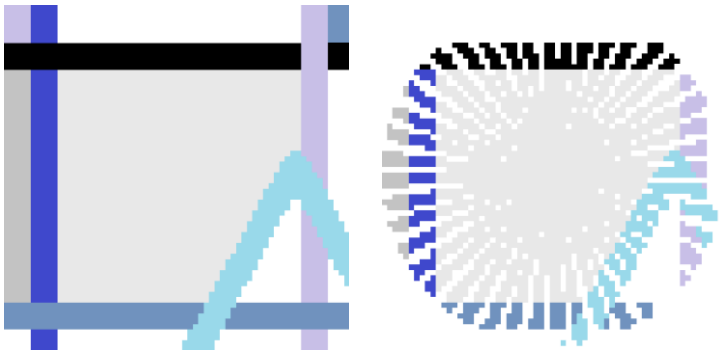

Figure 4-3. Image Converter Block test via Python. Left depicts the original image, right indicates recreated image with sampling every 2 degrees.

4.2.4. AXI Master

Once the output valid signal is set, the AXI master proceeds to send over the output map to the LED driver block. It sends the entire register by bit shifting and keeping track of the number of bitshifts until the entire register is sent to the LED driver, which also receives the image map in the same manner. Additionally, this block will set the a bit in the “config space” of the LED driver to indicate write complete.

4.3. LED Driver

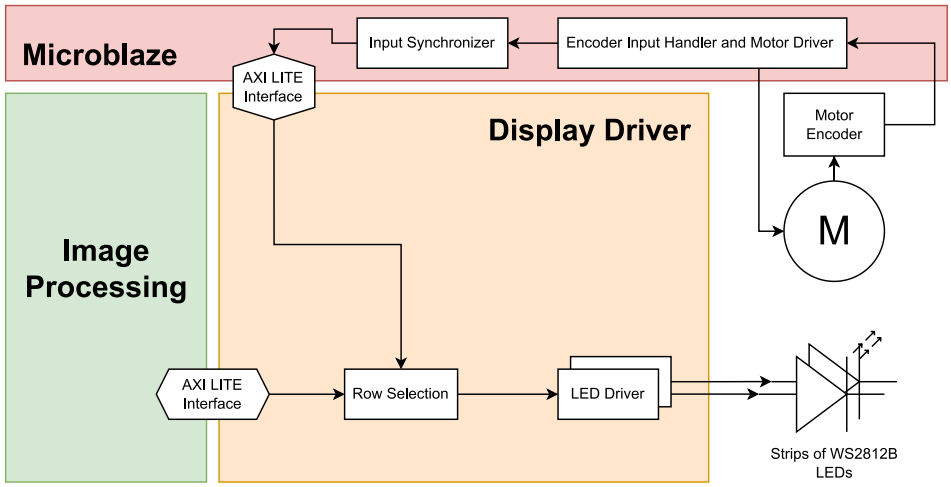

Figure 4-4: High level block diagram of the display driver.

The LED driver consists of two discrete blocks. Both are completely custom IPs. There is the LED driver block, used to generate the WS2812B LED data protocol, and a Row Selection block that selects a row of LEDs in the image to input into the LED driver.

4.3.1. LED Driver

This protocol specifies a 24-bit color encoding for each LED. The data is transferred over a single data line with no clock line, and the LED values are assigned sequentially, i.e. the first 24 bits of data is assigned to LED 0, and the second 24 bits of data assigned to LED 1 etc.

With no clock forwarding, a bit is denoted by the data line being raised and subsequently lowered. To send a 0 bit, the data line is held high for $0.4 \pm 0.150\ \mu s$ and held low for $0.85 \pm 0.150\ \mu s$. To send a 1 bit, the data line is held high for $0.8 \pm 0.150\ \mu s$ and held low for $0.45 \pm 0.150\ \mu s$. To reset all LEDs, the data line must be held low for $> 50\ \mu s$.

A custom IP block was created to generate this data signal. It is effectively a state machine that counts the number of cycles of the input clock and raises and lowers the data line. At an input of clock of $100\ MHz$, to send a 0 bit the data line is held high for 400 cycles and held low for 800 cycles. To send a 1 bit, the data line is held high for 800 cycles and held low for 400 cycles. These values were parameterized for ease of modification at the block level.

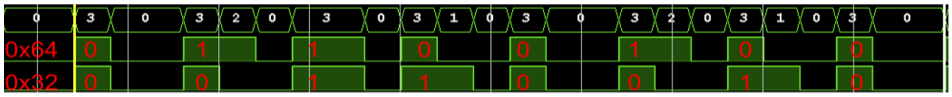

Figure 4-5: Simulated waveform of the implemented data tx for the WS2812B protocol.

During development, there was significant difficulties related to high LUT usage. Hence, the decision was made to encode each LED value as a 6-bit color scheme. Additionally, special care was taken to utilize bit shift logic instead of muxes to maximise FF usage instead of LUTs.

The input to this block are the clock and reset signals, and a $6 * Number\ LEDs\ per\ Strip$ bit LED values array. The output of this block is a single data tx wire which are constrained to the GPIO pins during implementation.

It should be noted that even with 6-bits per LED as an input, the actual protocol still requires 24-bits per LED. Hence, each 2-bit color value (3 colors) is mapped to 4 levels of brightness. These values are also parameterized for easy change in the block diagram.

Upon a reset signal, the input LED values are latched by a shift register. A reset counter is started to keep the data line low for 50k cycles ($50\mu s$). The driver then steps through the data for every LED, shifting the data down after each transmission.

4.3.2. Row Selector

Similarly, to the LED driver, one of the most critical design considerations for the row selector was reduction of LUT utility.

This block was originally designed to utilize ping-pong buffers to achieve simultaneous read and write capabilities. But this design proved to utilize too many LUTs. The design was therefore re-worked to use FIFO queues and shift registers.

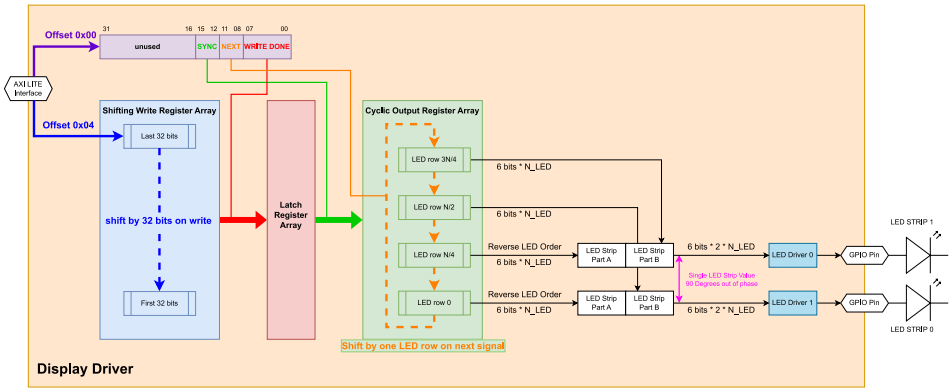

Figure 4-6: Detailed block diagram of the row selector component of the display driver

The display driver interacts with the rest of the system via a single AXI lite slave interface. This interface exposes 2 32-bit write registers.

The first register at offset 0x00 is the 'configuration space'. This is a register that is written to by upstream devices to control the behavior of the driver. The reason for this configuration space instead of just having an input wire was to allow fast debugging of the block. It was convenient to have the capability of testing block purely via the microblaze acting as the master. This way, there was no need to constrain external button pins to these signals or figure out order of button pushes for testing, only

changes in the C code were required. It was also convenient as all other blocks in the system required an AXI interface, meaning that integration would theoretically be painless as no modifications would need to be made.

The second register at offset 0x04 is the write register. To reduce LUT usage, the data written is not addressable directly. Instead, after each write, the data in the write register is shifted down to LSB by 32 bits. It was the responsibility of the image processing block to make sure the current number of bytes were being written.

When the image processing block finishes writing to the write register, it is responsible for sending the write done signal to the configuration space. This will inform the driver that the write register can now be latched and saved via the latch register.

The microblaze/encoder system is responsible for writing the next and sync signals. The encoder should send the next signal at regular intervals of theta. The encoder should send the sync signal when it completes a full rotation.

Upon a sync signal, the cyclic output register takes on the data in the latch register. This operation can only happen on a sync signal to ensure that the phase of the cyclic output register is aligned with that of the latch register.

Upon a next signal, the cyclic output register cyclically shifts down to LSB by the size of the row, where size of the row is 6 * Number of LEDs on a Single Arm. The output data wires to the LED drivers are statically assigned to regions of this cyclic register such that the shift down results in the next row of LEDs being displayed.

The image processing block outputs data in order such that each row in the order of LEDs from smallest radius to outermost radius. Each row thereby is a representation of one arm for a particular angle theta. However, since in the physical assembly, each LED arm was wired in series with the arm in the opposite phase, some concatenation of the rows is required to assemble the data for the entire strip.

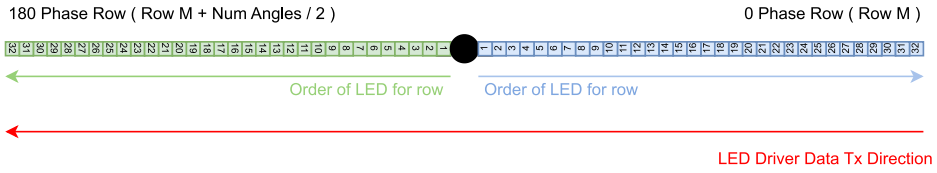

Figure 4-7: Diagram of an LED arm, with assigned row and indexes for LED values

What this effectively means for the output is that the 0^{th} row of the cyclic output register is reversed to satisfy the Tx LED order, and is subsequently combined with a non-reversed N/2th row of the cyclic output register, where N is the number of angles (rows) in the array. The same logic applies for the other two arms that exist perpendicularly.

4.4. Microblaze

4.4.1. Image Map Generator

This piece of C-code generates a static mapping of cartesian coordinates (pixel locations) to polar coordinates (LED locations around the display while spinning). It takes in the number of LEDs per arm and the number of updates-per-revolution, referred to as 'angles' at other points in this report. For every LED-angle combination it outputs the closest (x,y) coordinate in pixels. Each LED-angle combination can also be thought of as a polar coordinate, with this piece of code assigning the nearest cartesian coordinate (x,y) to each polar coordinate (r, theta). This static mapping of cartesian coordinates to polar coordinates would then be written into a BRAM segment accessible by the Image Processing block, had we fully finished our planned integration.

4.4.2. Encoder

Our team chose to use an AS5600 magnetic encoder to detect the position of the spinning LED display, allowing us to update the LED strips at the correct angle regardless of the speed at which the display is spinning. A small, diametrically polarized magnet (north and south pole split by the diameter, as opposed to split between the faces) is attached to the end of the motor's shaft and the AS5600 IC, which accurately senses the orientation of the magnetic field, is placed in parallel to the surface of this magnet. As the motor rotates, the changing magnetic field is converted to a 12-bit digital value by an analog-to-digital converter in the IC. This 12-bit value can be read using the I2C protocol by external devices, such as the FPGA. We utilized Xilinx's pre-canned AXI IIC IP, which comes preinstalled in Vivado, to do so. We set the bus speed of the IP to 800kbps, the highest speed compatible with the AS5600 encoder, to minimize the delay between requesting and receiving data. To set up the device, a status register is read from, indicating if the position of the magnet is acceptable. Reading value 55 (decimal) from the status register means the position is good. Once this value is returned, we continuously poll the two 'raw angle' registers and use bitwise operations to concatenate the 8-bit and 4-bit values into a raw angle value. This raw angle is then normalized upon every read by subtracting the initial value read from the encoder at startup. Upon each full revolution, an AXI write is performed to a specific address mapped to a register in the display driver, signalling it to 'synchronize' at zero degrees. This triggers the shift register in the display driver to begin filling with new data. After every (360.0 / NUM_ANGLES) (a parameter of the display driver IP) degrees of rotation, as detected by the encoder, a 'next angle' bit is written into this same address space to empty the bottom row of the shift register onto the LED strip.

4.5. Physical Components

4.5.1. LED Arms

The LED arms are 4 strips of high density WS2812B LEDs arranged in a cross pattern. They are housed in aluminum LED channels which also provide wire management.

4.5.2. Motor

The motor is controlled by a high-power (12V, 30A) motor driver which delivers power via the output of an h-bridge circuit. The output of this h-bridge is modulated via a PWM signal from an Arduino Nano. The original plan was to use the PmodDHB1 IP from Xilinx's vivado-library IP repository but there were compatibility issues with our selected motor driver. This IP from Xilinx was designed to work with the low-power HB1 PMOD (3.3V, 2A). The maximum PWM frequency it can output is 1KHz yet our high-

power motor driver required a PWM frequency of 16KHz. Furthermore, the PMOD pins on the FPGA are 3.3V logic but the motor controller requires 5V logic. We attempted to solve this voltage discrepancy with an off-the-shelf logic-level converter, but had no success, likely due to the PWM frequency being too low for our motor driver. We were therefore forced to interface the motor controller with an Arduino which outputs a constant PWM signal, causing the LED arm to spin at a constant speed.

Commented [ET1]: Just wrote what I remember from the presentation which could be wrong

4.5.3. Slip Ring Design

One critical aspect of our project was designing a slip ring in order to allow for the LEDs to be connected to power and the FPGA without having cables get tangled as the LED rods were spinning. We decided to use copper tubes and motor brushes to design a custom slip ring. The motor was connected to a mount on which the copper tubes would be placed, and at the end of the mount would be where we attach the LED rods. This entire component would be spinning. The motor brushes would be placed in mounts we design and attached to the base plate. The cables from the FPGA as well as power would be connected to the motor brushes, and the signal would be transmitted via the copper tubes, to the LED rods via cables soldered to the inside of the copper tubes. A model of this design can be seen below.

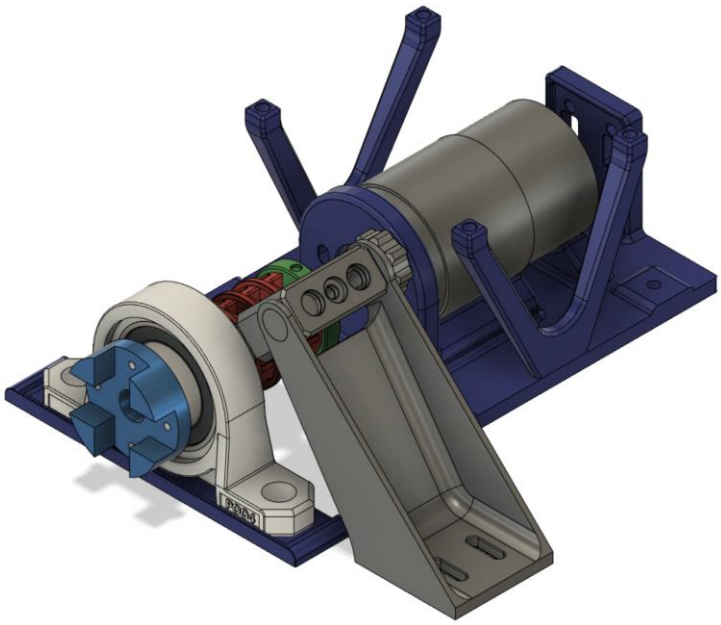

Figure 4-8. Final assembled design. Only one motor brush holder visible in this rendering; another will be placed on the other side.

4.5.4. Metal Core Component

This component became incredibly useful to have when trying to decide how to assemble the slip ring. We decided that as opposed to having multiple couplings, it would be simplest to have one piece of

metal attached to the motor as well as hold the LED strip chassis. Eddie thus designed and machined the metal core component, which can be seen in the figures below.

The LED arm mount attaches the LED arms to a shelf and provides a rotating electrical interface to send data and power to the LEDs. An adapter was machined out of aluminum to hold the 4 LED channels and attaches them to a pillow bearing and a motor via a flexible coupling. The adapter also houses 4 copper rings which have wires extending from them through the adapter and to the LED strips. These copper rings interface with 4 motor brushes which are mounted on the shelf adjacent to the adapter. These motor brushes connect to the FPGA which sends the data signals to the LEDs, and to a power supply for the LEDs.

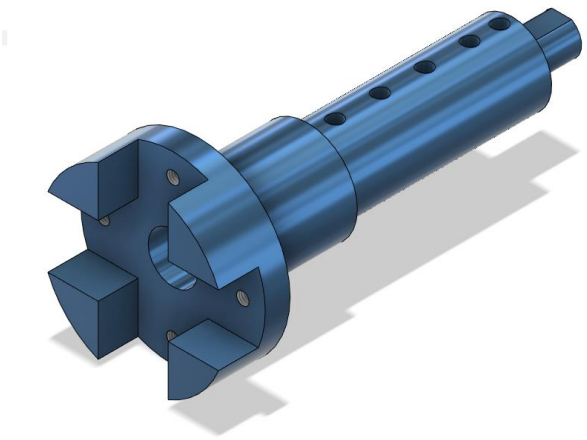

Figure 4-9. Metal core piece as rendered in Fusion360

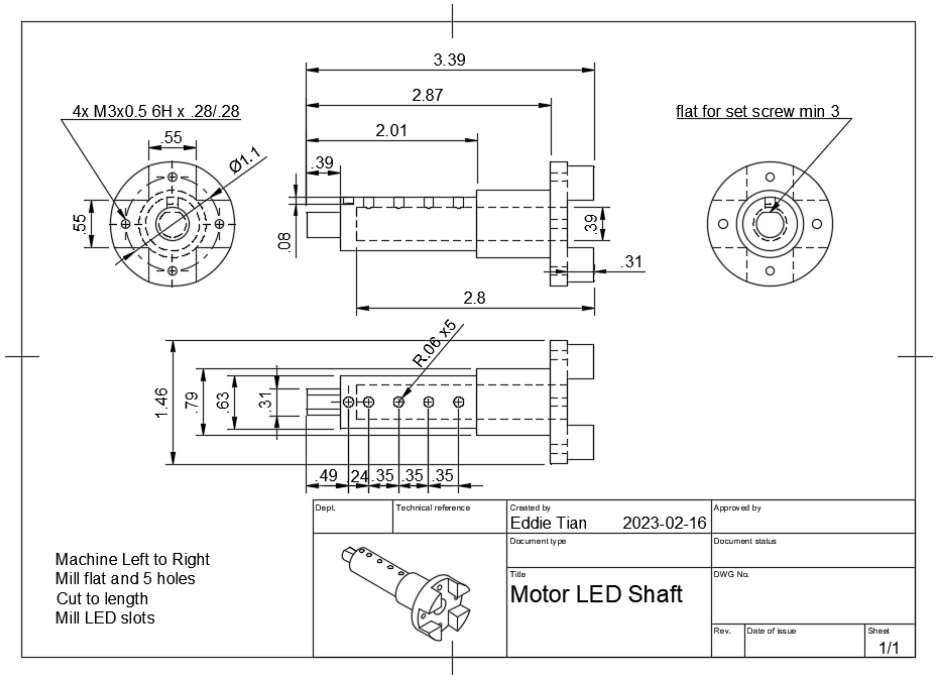

Figure 4-10. Metal core piece specifications for machining.

5. Description of Design Tree

Our project can be found here: https://github.com/CarbonicKevin/G3_SpinningLEDDisplay

The design tree can be summarized as follows:

- **docs**
  - device_manuals: various manuals for the different devices used
    - nexys-video_rm.pdf
    - WS2812B.pdf
  - group_submissions: contains the final report and presentation
- **test**: contains test scripts for coordinate mapper algorithm
  - test.py: test script to test cartesian to polar mapping algorithm
  - test.c: script to generate coordinate map, to be integrated with microblaze
  - test_image2.png: test image for the coordinate mapper
  - test_image2_o.png: output of test for coordinate mapper with 180 update angle intervals
- **srcs**
  - ip_repo: exported custom IPs
    - display_driver: display driver repo
    - c2p_proj: cartesian to polar mapping repo
    - sd_card: sd card reader repo
    - digilent_precanned: digilent IP repo for Vivado found here
  - projects: folders that contain files for various projects
    - display_driver: projects to test the display driver
      - scripts
        - led_values_gen.py: script to generate test LED values
      - src_microblaze: test code folder for running with project_microblaze.tcl
      - src_sim: contains simulation files
      - project_microblaze.tcl
      - project_vip_sim.tcl
    - c2p_proj: projects to test the cartesian to polar mapping block
      - src_sim
        - car2pol_sim.sv: simulation file for sd_c2p_integration.tcl
      - sd_c2p_integration.tcl
  - integration
    - sd_c2p_integration
      - src_sim: simulation sv file for sd_c2p_integration.tcl
      - sd_c2p_integration.tcl: tcl file to recreate integration block diagram
    - display_driver_and_microblaze
      - tcl_script: tcl file to recreate display driver integrated with encoder reading block diagram
      - vitis_src: SDK source files (C-code)

- integration.srcs\constrs_1\new\integration.xdc: constraints file binding PMOD port for external connection to encoder and h-bridge

6. Appendix

6.1. Project Schedule

6.1.1. Week 4: 30th January – Milestone 1

- Lab Date: **February 2nd**
- Define parameters for hardware blocks
    - Block inputs and outputs
    - Data Bandwidth Requirements
    - Memory Requirements
    - Interface methods with other blocks
- Assign hardware blocks to individual team members.
    - Each member should at least generate a diagram or plan for a block.
        - Blocks like the coordinate conversion block may take longer to implement.
- Pick materials and devices required for physical rig and begin procurement.

6.1.2. Week 5: 6th February – Milestone 2 & Proposal Presentations

- Lab Date: **February 9th**
- Modeled slipring prototype with CAD
- Assembled ring mount and motor mount
- Started on LED driver IP
- Looked into SD card IPs
- Have Arduino code to spin motor and assign LED values

6.1.3. Week 6: 13th February – Milestone 3

- Lab date: **February 16th**
- Working on a second iteration of the physical prototype
- Started on image conversion block by doing cartesian to polar mapping in Python
- Started on enclosure design for the prototype
- Looking into HDMI, found some IPs
- Designed the metal core component
- Encoder I2C not working
- Motor driver working

6.1.4. Week 7/8: 20th February – Reading Week and Mid Project Demo

- Continued to work on the physical prototype and improve the design

6.1.5. Week 9: March 6th – Milestone 4 & Mid-Project Presentations

- Lab date: **March 9th**
- Assembled and soldered physical model, metal core machined and integrated
- LED driver has AXI-lite based double register implemented and integrated with Microblaze system

- Image mapper in Python complete. Starting to convert map generating code to C and conversion algorithm to Verilog
- Implemented SD IP for regular SD cards and wrote SD card IP caller to read pixels from the SD card and place them in memory for the microblaze to read.
- Encoder I2C still problematic

6.1.6. Week 10: March 13th – Milestone 5

- Lab date: **March 16th**
- Working on assembly of final construction, working with single LED strip for now
- LED driver reconfigured to use FIFO queues for reduce LUT requirements
- Started on AXI interface of the image transformation block, currently debugging.
- Cabinet assembly completed
- SD card reader challenging, changed to different IP using VHDL instead of Verilog. Developed FSMs for SD card operations.
- Encoder I2C finally working, basic API written for setting up encoder and reading a raw angle

6.1.7. Week 11: March 20th – Milestone 6

- Working on AXI interface between SD card and image transformation block
- Debugging SD card IP
- Image mapping block AXI mostly working

6.1.8. Week 12: March 27th – Final Demo

- Attempt to integrate image mapping and SD card block
- Attempt to integrate the two previous blocks to the project