Digital Sundial Final Project Report

Moaaz Fayumy
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
moaaz@mit.edu

Asaad Mohammedsaleh
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
asaadm@mit.edu

Hamza Raniwala
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
raniwala@mit.edu

***Abstract*—We present a design for a digital sundial which processes a sundial signal viewed under ArduCam and presents the time of day via angle calculation signal processing, as well as triggering an alarm clock with SD-card programmed alarm outputs at specific solar times. The hardware consists of a Urbana board with Xilinx Spartan7 FPGA, Arducam, stereo speaker, and micro-SD memory.**

***Index Terms*—solar angle, lookup table, discrete cosine transform,**

I. Hardware Setup

A Xilinx Spartan 7 FPGA interfaces with the Arducam Mini and supplies an HDMI-clocked signal to an edge detection and center of mass module. These are used to calculate the clock angle and therefore current time, which in turn reaches an alarm clock module.

We use Arducam Mini 2MP Plus OV2640 camera that can support up to 1600x1200 images. A micro-controller to interface between the camera and the FPGA. Audio data is stored on a 2GB MicroSD card, which is read and then output through a stereo PWM audio speaker.

II. Image Processing: Sundial Angle Calculation (Hamza)

An image processing algorithm that compares the displaced center of mass of a sundial (where pixels are "turned off" by the sundial shadow) to the sundial's true center (determined by edge detection of thresholded pixels on the edge of the sundial) to determine the shadow angle and length. This information allows for calculating the time of day. Consequently, the digital sundial math falls into two parts: (1) implementing a trigonometric calculation to determine the angle of the sundial shadow and (2) implementing a length calculation of the shadow length via the quadratic formula.

A. Algorithm Implementation in OpenCV

Fig.1 demonstrates a pre-hardware implementation of the algorithm using the OpenCV package in Python to calculate the angle and length of a shadow on a simulated thresholded sundial image. The OpenCV implementation consists of four steps outlines below.

First, the simulation image is threshold masked by pixel brightness. Pixels representing a sundial are assigned a value of 1, whereas pixels around the sundial and within the sundial shadow are assigned a value of 0.

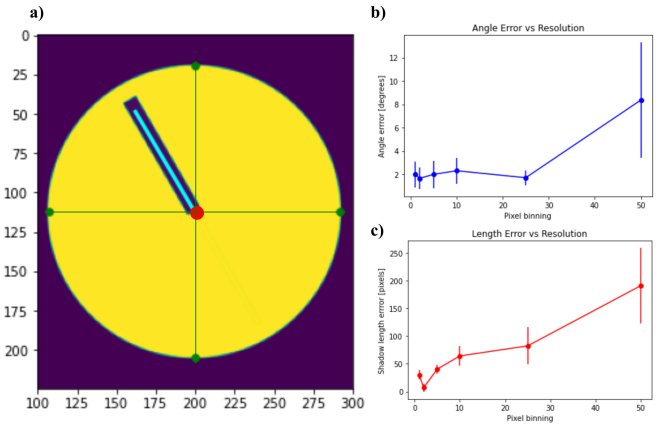

Fig. 1. Implementation of shadow calculations in OpenCV. (a) Display of the sundial with overlaid image processing. The green cross hairs target the true center of the sundial determined from edge detection. The red circle corresponds to the shadow-shifted center of mass. The blue line shows the calculated shadow angle and length. (b) Absolute error of shadow angle as a function of pixel binning (i.e. inverse resolution). (c) Absolute error of shadow length as a function of pixel binning.

Second, the center of mass of the sundial is calculated using the formulae

$$x_{CoM} = \frac{\sum_i x_i}{\sum_i i}, \tag{1}$$

$$y_{CoM} = \frac{\sum_i y_i}{\sum_i i}. \tag{2}$$

The true center of the sundial is calculated by comparing the edges of the sundial as

$$x_{cen} = (\max(\{x_i\}) + \min(\{x_i\}))/2, \tag{3}$$

$$y_{cen} = (\max(\{y_i\}) + \min(\{y_i\}))/2. \tag{4}$$

Next, the angle of the sundial shadow is determined as

$$\theta = \text{sign}(y_{CoM} - y_{cen}) \times \arccos\left(\frac{x_{CoM} - x_{cen}}{\sqrt{(x_{CoM} - x_{cen})^2 + (y_{CoM} - y_{cen})^2}}\right). \tag{5}$$

If we assume that the shape of the sundial is roughly ellipsoidal and the shape of the gnomon (sundial shadow)

is roughly rectangular with width w, then we can approximate the shadow length using the quadratic formula with the length of the center displacement $r = \sqrt{(x_{CoM} - x_{cen})^2 + (y_{CoM} - y_{cen})^2}$ and the ellipsoid area or "pixel mass" of the sundial $m = \pi ab$ as

$$L = \frac{-wr + \sqrt{(wr)^2 + 2wrm}}{2w}. \tag{6}$$

The accuracy of this method in OpenCV increases with pixel resolution, as shown in Fig.1(b,c), motivating the use of a high-pixel camera. The current hardware implementation of the edge detection and center of mass modules is shown in Fig.3.

B. Trigonometry Lookup Module via Python

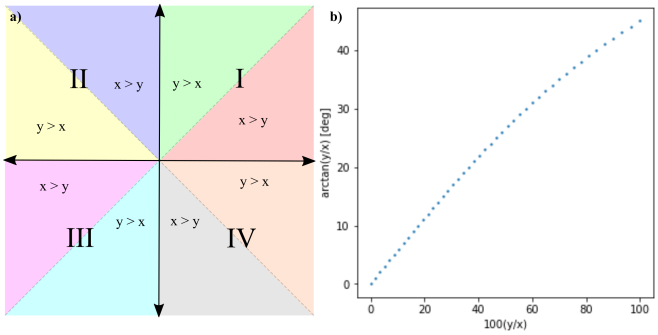

Fig. 2. (a) Identification of quadrants and $x > y$ or $x < y$ regions on the Cartesian plane. (b) Discretization of the 0-45 degree angle range as a function of $\arctan(y/x)$ against $100(y/x)$ to generate elements of a lookup table (LUT).

Implementing the pristine arccosine function in SystemVerilog would require an IP as well as the use of real numbers, which are slow in calculation compared to logical numbers, especially when full resolution on the arccosine output is not needed to achieve minutes-scale resolution on the sundial angle. Therefore, we implement an angle-finding module that can compute an angle given input $\{x_{CoM}, y_{CoM}, x_{cen}, y_{cen}\}$ in 1-100 cycles for 45-degree resolution. The algorithm does the following:

1) Decide which quadrant of the Cartesian plane the value pair $x = x_{CoM} - x_{cen}, y = y_{CoM} - y_{cen}$ falls in, as well as if $x > y$ or vice versa;
2) If $x > y$, then x and y are switched for step 3 and 4.
3) Calculate the ratio $R = 100y/x$, where y and x are determined by which eighth of the Cartesian plane one is in via truth table (Note that the ratio y/x is multiplied by 100 so that 45-degree resolution can be achieved with the ratio output);
4) Use a LUT to determine an angle θ_{init} from 0 to 45 degrees based on R;
5) Output the final sun angle via truth table using which-quadrant information and $x > y$ or vice versa using the formula

$$\theta_{final} = \begin{cases} 90(i-1) + \theta_{init}, & x > y \\ 90(i) - \theta_{init}, & y > x, \end{cases} \tag{7}$$

where i is the quadrant number.

The full algorithm outlined above will be included in the angle_division.sv module found in the final project submission.

We use a Python script to generate the SystemVerilog code for an angle LUT with 45-degree resolution on the LUT output which outputs an angle from 0 to 45 degrees based on the ratio between $(x_{CoM} - x_{cen})$ and $(y_{CoM} - y_{cen})$. The range from $\tan(0)$ to $\tan(1)$ is discretized in 46 steps and correlated in a case-based LUT to the output of the ratio $100((y_{CoM} - y_{cen})/(x_{CoM} - x_{cen})$. A ratio multiplier of 100 is sufficient to achieve 45-degree resolution in one eighth of the Cartesian plane, as no duplicate truth table cases are found when rounding the output R to the nearest whole number. (i.e. for every output R, there is a unique angle out of 45 degrees).

We testbenched the above approach for each Cartesian quadrant, partially shown in Fig.3(right). The use of $x > y$ truth values allows us to restrict the timing of the angle_division.sv module by allowing us to, at most, divide $100y/x$ in the worst-case scenario where y equals x–which only requires 100 clock cycles using the class's divider module. In the event that an input value $y_{CoM} - y_{cen}$ exceeds $x_{CoM} - x_{cen}$, x and y are flipped in the divider module before sending to the 45-degree-resolution LUT. This means that all angle calculations conclude within ~ 100 clock cycles, which comfortably fits in the vertical blanking and sync period at the end of each HDMI frame. Thus, our angle calculation is easily completed in time for a new frame.

While our original aim was to determine an angle value from 0 to 120 degrees (the expected shadow angles throughout daytime), our final build supports full 360-degree angle calculations. The primary limitation of the angle module in its current form is the selected resolution. With a 45-degree-resolving LUT and quadrant / octant discretization, we can only output values with single-degree resolution, whereas to resolve single minutes of time on the sundial, we would need 3600 minute resolution. Increasing the LUT resolution by a factor of 10 (so we can resolve tenths of degrees) would remedy this issue at the cost of additional memory utilization (10x LUT usage) on the FPGA.

C. Quadratic Formula implementation for length calculation

The length calculation is completed using the pipeline in Figure N. Similarly to the angle calculation, we use a square root table lookup to skip conversion to real numbers and slow math implementations. The primary constraint is that this approach heavily utilizes LUTs on the Spartan 7 FPGA, leading to timing issues for higher resolution square root calculations. In order to avoid division with large numbers, which would take innumerable clock cycles, the length formula and was simplified to

$$L = \frac{r}{2}\left(-1 + \sqrt{1 + 2(\frac{m}{wr})}\right). \tag{8}$$

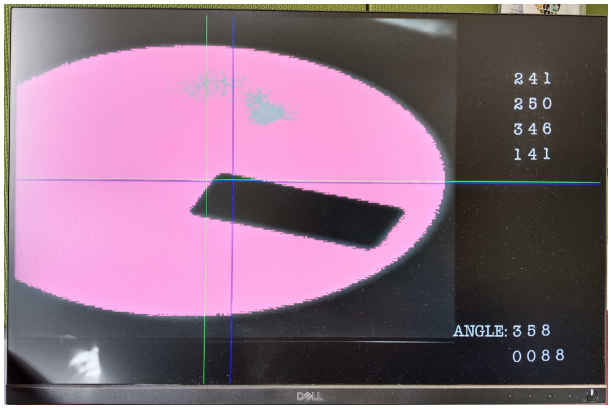

Fig. 3. Final build's HDMI output. The cross hairs indicate the center of mass (blue) and edge detection (green) module implementations, where the green cross hairs indicate the center of mass and the blue cross hairs indicate the edge-detected true center. The alarm clock values (top right) are generated using a number sprite sheet and the alarm values stored on the FPGA. The calculated angle from the sundial image is displayed in the bottom left, and the estimated length of the shadow (in pixels) is displayed directly underneath the angle.

The displayed length calculation depends on calibrating the expected width of the shadow to read accurately. As the camera positioning above the sundial changes, this modifies the actual width of the sundial shadow. Hence, the length calculation, while implementing the math found in Equation (8), does not currently output an accurate length estimate in pixels. Given that our build requires touching the FPGA to change the alarm clock controls, this is currently a limitation of the design and can be circumvented by extending the camera wires away from the body of the FPGA and fixing the camera position relative to the sundial signal.

The LUT approach to angle and square root calculations, while fast in terms of clock cycles, consumes the most memory on the FPGA. In the final build, over 5000 LUTs were utilized, where the angles LUT and value in top_level.sv occupied a 256 x 6 LUT alone. Together with multiple square root LUTs, this became a memory-hungry implementation of the math pipeline, and we were forced to reduce the numerical resolution of the square root LUTs to build on the FPGA with its limited resources. While the math still finishes in time for a new frame, this is a suboptimal result on the FPGA. In future optimizations, each step of the length calculation should be sent to an independent multiply module wtih AXI protocol or other handshake mechanism to prevent large operations from being forced to run in single clock cycles.

III. Visual display of sundial (Hamza, Moaaz, Asaad)

The final HDMI output appears as shown in Figure 3. Two crosshairs identify the center of mass (green) and the true center (blue) of the sundial, respectively, and the calculated angle and length are displayed in the bottom right corner of the screen using a number sprite sheet. The current alarm "angles" are reported in the top right of the screen. These communicate with alarm values set on the FPGA (discussed in the audio and memory section), allowing us to display the currently stored alarm values. In order to avoid failing timing requirements on the FPGA, a double dabble formula was used to calculate the length decimal places for visual output; the angle decimal places were able to be calculated via modulo equations without failing timing.

The HDMI pipeline was left mostly untouched, with the primary modification occuring in the video mux module which now handles an additional crosshair as well as several word and number sprites. We extracted an additional 24.75 MHz clock from the HDMI clock wizard module to handle audio timing.

IV. Camera Pipeline (Asaad)

We opted to use the Arducam OV2640 camera as a substitute for the OV7670 camera used in labs 5-6, aiming to achieve higher accuracy in the image processing modules by feeding them a larger number of pixels. The Arducam camera, similar to the lab camera, has a default resolution of 320x240 RGB (5-6-5). To obtain a different resolution, the Arducam can provide it in JPEG compressed form. In this section, we will go through the decompression algorithm, attempts to include it in our system, and the state of the camera at the time of writing this report.

A. JPEG Decompression

The JPEG decompression algorithm comprises four main stages: Huffman decoding, de-zigzagging and de-quantization, reversing Discrete Cosine Transform, and finally, transforming the color space from YCrCb to RGB. JPEG images are stored in the format illustrated in Fig. 4. To implement the algorithm into an FPGA

Header information
Quantization table (luminance)
Quantization table (chrominance)
Start of frame information
Huffman table (luminance - DC)
Huffman table (luminance - AC)
Huffman table (chrominance - DC)
Huffman table (chrominance - AC)
Start of scan information
Coded image data

Fig. 4. JPEG image format [1]

system, we explored a reference implementation found at https://github.com/ultraembedded/core_jpeg/tree/main. The setup part of decompression involves reading and storing six tables: two quantization tables of fixed 8x8 sizes and four Huffman-encoding tables of varying sizes depending on the captured image. The next step is to read the compressed data and decode them back into RGB values. Due to the design of the compression algorithm, decompressing processes 8x8 blocks (minimum coding unit, MCU) at a time. Therefore, from other FPGA modules' perspective, they need to receive 8 x width of the image to get the complete first row for

the HDMI pipeline. This requires buffer memory to store decompressed rows before passing them to the video pipeline. However, the Arducam provides 4:2:0 subsampled captured JPEG images, which results in 16x16 MCU sizes instead of the standard 8x8, requiring more buffer storage for the image. When the FPGA receives compressed image data, it needs to look up values from the provided tables. This was an apparent issue during the testbench of the reference code.

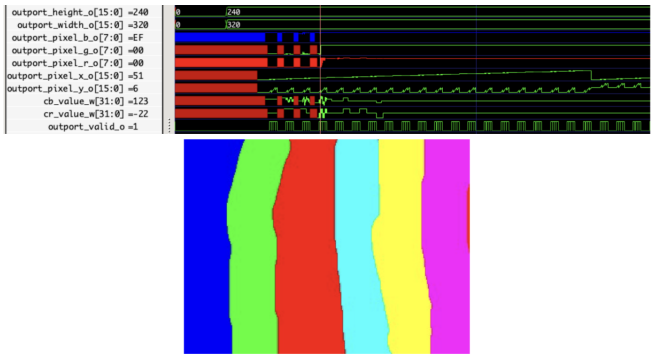

Fig. 5. Top is testbenching run of the colors image below. The region were outputs are getting out correctly are in the blue region, but before the end of the blue region, the module stops producing correct outputs. It can be seen that the module is outputting valid output pixel signals (outport_valid_o) but no readings of YCbCr are happening at the same time.

As can be seen from the testbench screenshot, decompressed data comes in groups of four chunks, where each chunk corresponds to an 8x8 block of the original image. The issue encountered was that colors and YCbCr values stopped changing after a certain point, and we were unable to debug the reference module further.

Due to time constraints, we attempted to implement decompression in the microcontroller by modifying reference C code available in the same repository as the JPEG decompression reference module, but we could not complete it by the deadline. The challenge here was to modify the code to handle a stream of data instead of accessing an array of stored data, which proved difficult to finalize before the deadline.

B. Camera-Microcontroller Setup

To connect the Arducam with the FPGA, we used an ESP32 microcontroller as an intermediary device that handles communication with the camera and forwards the image bytes to the FPGA, see Fig. 6. The Arducam supports up to an 8 MHz clock speed for SPI communication, so approximately, it sends a 16-bit RGB pixel at a 0.5 MHz clock rate. The ESP32 then transfers these bytes to the FPGA at approximately the same speed, but we observed slow drawing in implementation (3.4 seconds per frame). One factor contributing to this behavior is the use of digitalWrite() calls to set 8 pins of data when sending a byte to the FPGA.

We tested SPI communication between the ESP32 and the FPGA to achieve faster frame rates, but it did not significantly improve performance.

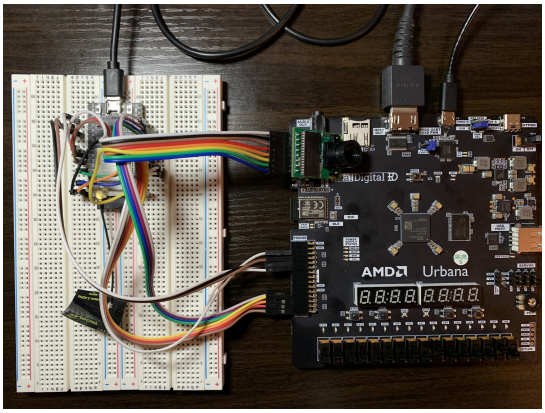

Fig. 6. Arducam OV2640 connected to ESP32 microcontroller through 8 pins (4 SPI pins, GND, VCC, SDA, and SCL). Microcontroller connected to FPGA through 13 wires (8 bits transmission, PMODBCLK, PMODBLOCK, CAM_CLK, HREF, and VSYNC).

V. Audio and Alarm System (Moaaz)

A. Overview

In this project, we have implemented an audio system capable of reading and playing multiple WAV audio files stored on an SD card. This approach was necessary due to the limitations in the memory size of our FPGA, which could not accommodate the large audio files directly.

B. SD Card Controller

In our design, we used an SD card controller module developed by Fischer Moseley for the Nexys DDR board [2]. We needed to test this module to ensure it worked with our new board. The module reads data at a rate of 8 bytes at a time in 512-byte blocks (sectors).

C. SD Card Storage and Data Buffering

The WAV audio files, composed of raw pulse-code-modulation (PCM) data, are stored on an SD card and accessed using the sd controller module. To play an audio file, the system initiates a 512-byte read from the SD card, transferring each byte one at a time to a 1024-byte buffer in the FPGA's RAM. This buffer is divided into two halves: the first half is filled with new data from the SD card, while the second half is used for audio playback. As shown in Fig. 7, the system starts by loading a known amount of audio data from the SD card into the first 512 Bytes in the buffer. As playback progresses, it simultaneously reads the next sector of audio data into the second 512 Bytes in the buffer. This setup ensures continuous, smooth playback with minimal buffer size, preventing any cuts in the audio.

D. Audio Processing

The raw PCM data from the SD card is processed for playback. A 48kHz clock signal is generated to match the audio's sampling rate. The audio data is then passed from the buffer through a 31-tap Finite Impulse Response (FIR) filter. The FIR31 module we made was inspired from the instructions

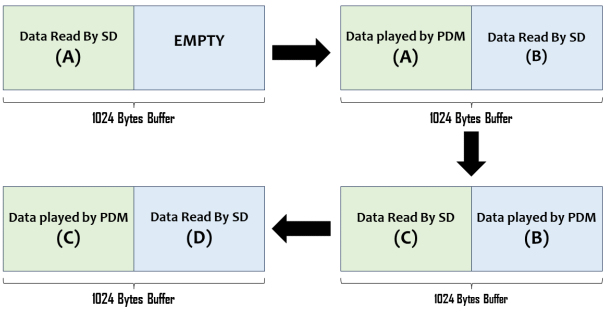

Fig. 7. Buffer Logic implementation

of a past class task in Fall 2019 [3], which can be used to filter frequencies at a certain cutoff frequency. The FIR filter's coefficients are set by a Python script based on the frequency response of each audio file, ensuring that only unwanted noise is removed while preserving the audio's main frequencies. Demonstration of the filter working is shown in the Fig. 8. After filtering, the audio signal undergoes analog-to-digital conversion through either a Pulse Width Modulation (PWM) or a Pulse Density Modulation (PDM) module, depending on user selection. This conversion is to make data ready to be played in speaker output.

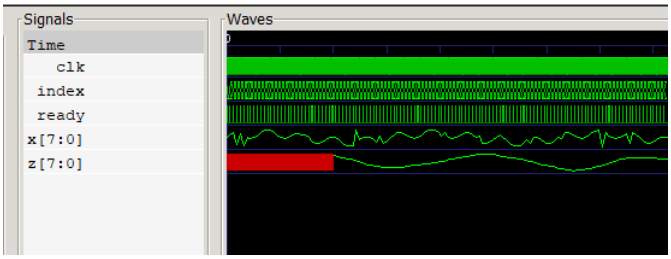

Fig. 8. Testbenching FIR filter. The input x is a noisy signal with sinusoidal behavior, and the output of the module is z sample.

E. User Interface and Control

The alarm system features a user-friendly interface on the FPGA. An eight-digit seven-segment display is used for interaction, where the upper four digits show the alarm time in a decimal format rather than showing hexadecimal which our display controller was built for (by modifying the logic of seven-segment-controller module). The lower four digits display the selected audio track number. We have four audio tracks uploaded to the SD card. The audio tracks can be cycled through using BTN1.

F. Setting and Managing Alarms

The alarm time is adjustable using BTN3 to increase the time by 10 and BTN2 to decrease by 1, allowing efficient navigation within a range of 0 to 360. To set an alarm and proceed to the next one (cycling between four alarms), both BTN2 and BTN3 are pressed simultaneously. This action saves the current alarm time and advances to the subsequent alarm setting. The current alarm status is indicated by two RGB lights on the board, with different light patterns representing which alarm you are setting. the Fig. 9 demonstrates the interface on the FPGA.

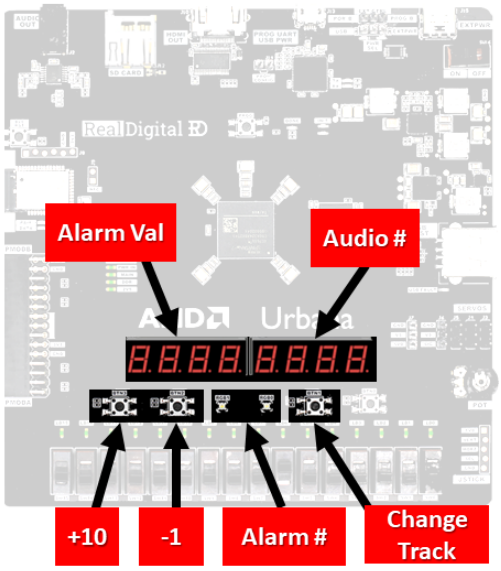

Fig. 9. Diagram of FPGA alarm clock interface. BTN1 cycles through audio tracks, BTN2 decreases it by 1, and BTN3 increases alarm time by 10.

G. Integration Overview

The integration of our audio and memory systems starts with turning on the SD card controller. This controller is set to read a specific audio file from a certain address. The audio data, which is in WAV format, is then sent into the dual-port RAM buffer. In this setup, one part of the buffer plays the current audio through the PDM module, while the other part is getting filled with the next piece of audio data. This keeps the audio playing continuously. The integration of the audio playback system with the alarm functionality provides a comprehensive alarm feature. Users can select different audio tracks, set multiple alarm times, apply FIR filtering, and choose modulation technique.

H. Testing and Validation of Audio Playback System

Our testing for the audio system was divided into three main parts. In the first part, we focused on testing the SD Card Controller. The objective here was to test the controller's ability to read data, and check that everything was working as we expected. We simulated a 25 MHz clock and controlled signals like miso, rd, and reset, which were implemented in the SPI protocol logic inside the controller, while observing the output data that we were reading. The results showed successful read operations with correct data capture and state machine transitions.

The second part of testing was the audio playback system test. The objective here was to test the audio playback process.

We simulated data coming from the SD card controller with the correct timing, and then tested the data being written to the dual-port RAM. After that, we observed the input of the PDM module and ensured it was what we expected it to be. The test bench results also showed correct behavior that we expected as shown in Fig. 10.

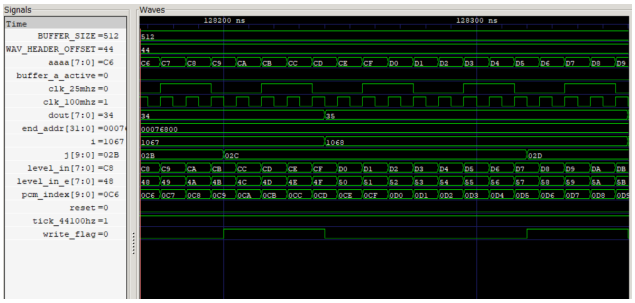

Fig. 10. Testbenching of Audio Playback System.

In the final stage, we physically tested the system on the FPGA using an Integrated Logic Analyzer (ILA) IP module in Vivado. This was necessary to ensure that the SD card's behavior aligns with our expectations from our simulation, specifically in terms of precise timing and correct logic. We observed, in real-time, read operations and buffer switches. The input to the PDM module (level-in) did follow the expected behavior.

VI. Conclusion

The delivered product for our digital sundial is able to position a camera above a sundial signal and extract the sundial shadow's angle and length with single-degree angle accuracy and uncalibrated length accuracy. The build successfully communicates this sundial signal to an alarm clock with four reconfigurable alarms and an alarm audio of choice read out from an SD card in WAV audio format. The final build has two versions–one which works with the camera from lab, and another which communicates with an ArduCam via byte-by-byte communication.

The angle can immediately be correlated to a clock time (or alarms can be set with the sundial angle itself, as in our final build), and parallax compensation was ultimately deemed unnecessary for the final build due to the required proximity of the camera to the sundial making parallax compensation a moot point. Determining the sun's elevation from the shadow length would require feeding the shadow length and a predefined comparison length variable into a second angle calculation module; this was not implemented in the current build. We did not include a PID loop to perform noise compensation on the output length, which is updated frame by frame in the final build and can glitch as a result. As such, we met a mixture of the commitment "Working Sundial Image Processing Pipeline" and initial components of the reach "Complete Sundial Image Processing Pipeline". However, we were still happy to complete these calculations with no use of IP.

In the development of the alarm clock pipeline, we successfully achieved the following commitments. First, the utilization of SD card-uploaded WAV Audio File for alarm sound: We effectively implemented a system where WAV audio files uploaded to an SD card serve as the sound for the alarm. Secondly, the Graphical interface for alarm setting on FPGA: A user-friendly graphical interface was developed on the FPGA for setting the alarm. This interface made it convenient to interact with the alarm system.

Additionally, we reached our goal for the complete audio pipeline: the incorporation of an audio effect - FIR (Finite Impulse Response) filter: the FIR filter enhanced the audio quality, allowing us to selectively filter frequencies and reduce unwanted noise in the alarm sounds.

Overall, this project indeed allowed us to explore interesting use cases of an FPGA. We were able to learn about automated scripting of SystemVerilog code via Python, handling SD card memory and WAV audio files, standard JPEG compression/decompression algorithms, and interfacing between micro-controller software and FPGA hardware. While some of these components made it to the final build and others didn't, we found this final project to be a greatly educational experience with digital systems. In future builds, we would dedicate more time up front to understanding the hardware to which our FPGA needs to interface so we can more easily distinguish problems with electronics from bugs in our implementation.

VII. Code Repository and Block Diagram

The code repository of the final project is located at https://github.com/hraniwala/6_205_Final_Project_Digital_Sundial. Figure 11 shows the block diagram of the final project.

VIII. Contributions and Acknowledgment

Asaad developed three interfaces for the ArduCam for this build, including Python testing scripts, byte-by-byte communication, and a testbenched JPEG signal-to-decompression pipeline. Moaaz completed the audio interface, SD card reader, alarm clock for the FPGA with some assistance from Asaad. Hamza wrote the sundial calculation modules, the visual display, and the visuals-to-alarm clock communication in the final build.

We extend many thanks to Dr. Joe Steinmeyer, who taught us and supervised our final project, as well as the many TAs and LAs who assisted us with the final project and throughout the term.

References

[1] Y. Khalid, "Understanding and Decoding a JPEG Image using Python," Yasoob.me, Jul. 14, 2020. [Online]. Available: https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/. [Accessed: Nov. 22, 2023].

[2] F. Moseley, "Starter Designs for [6205/sd-card]", GitHub repository, Available: https://github.mit.edu/6205/. [Accessed: Nov. 22, 2023].

[3] "Lab 5: Audio," Massachusetts Institute of Technology, 6.111 Introductory Digital Systems Laboratory, Fall 2019. [Online]. Available: https://web.mit.edu/6.111/volume2/www/f2019/handouts/labs/lab5_19a/. [Accessed: Dec. 13, 2023].

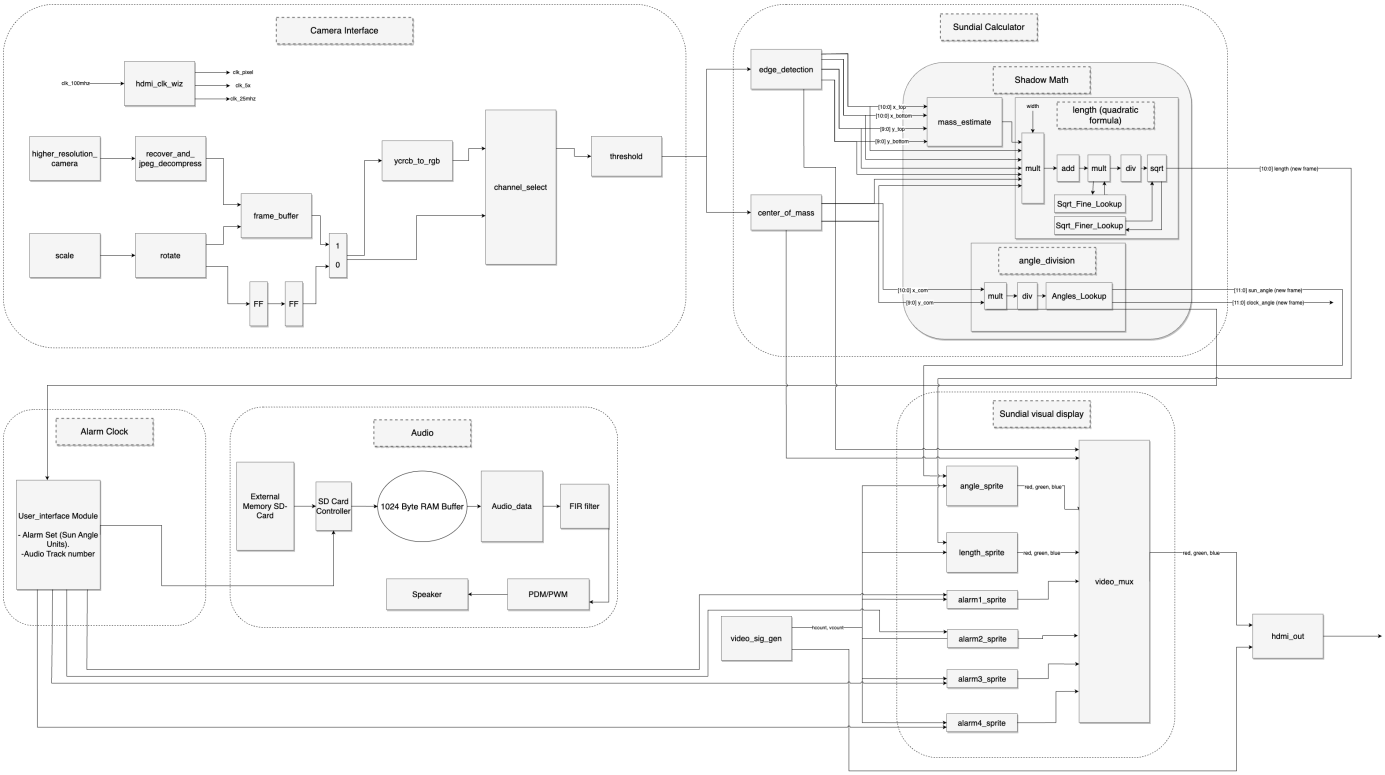

Fig. 11. Block diagram of the final build.