FPGAceTime Final Report

Sanjith Udupa*, Annie Wang*
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
sanjithu@mit.edu, awang27@mit.edu

***Abstract*—We present FPGAceTime, a bidirectional live audio/video communication system implemented across two RealDigital Urbana FPGAs. Each FPGA captures camera data, downsamples to a 320×180 RGB565 frame with anti-aliasing filters, captures audio from a microphone, and constructs interleaved audio and video packets serialized across a 20 MHz differential PMOD link. At the same time, each receives audio and video from the other board, and composes the final display and audio via HDMI and PWM audio output. The communication channel uses a custom 18-bit packetized protocol running over TMDS_33 differential pairs driven by OBUFDS/IBUFDS primitives with an external 220 Ω pull-up network. The end result is full bidirectional 320×180 RGB video and continuous audio transmission at about 17.5 frames per second. In this report we describe the system architecture, transmit/receive state machines, serializer/deserializer design, memory organization, and evaluation results.**



Fig. 1: Working FPGAceTime. Two FPGAs are connected via PMOD B cables, with HDMI outputs displaying remote video feeds and local selfie overlays in the bottom right corners on two monitors.

I. Introduction

FPGAceTime is a bidirectional, low-bandwidth, real-time audiovisual link implemented on two identical FPGAs. Each board captures live camera frames, performs anti-aliased downsampling for bandwidth reduction, packetizes video and audio data into fixed-format packets, transmits these packets over an FPGA-to-FPGA serial link, receives and decodes packets from the remote board, and composes remote video with a local selfie-view overlay. The system achieves approximately 17.5 FPS with 320×180 video resolution and 8 kHz audio, exceeding our original 10 FPS target and reaching our stretch goals for continuous audio and video. Figure 1 shows the working video part of our system, where each monitor displays a large remote video feed with a smaller selfie overlay in the lower right corner. Figure 7 shows our system block diagram, which we explain throughout this report.

A major design constraint is the limited number of free I/O pins on the Urbana board once the camera is attached. The OV5640 camera uses the entire PMOD A header. Only the PMOD B header is available for board-to-board communication, giving eight pins total. To maximize usable bandwidth over this narrow interface, we implement differential signaling using the `OBUFDS` and `IBUFDS` primitives. The corresponding pins are assigned the `TMDS_33` I/O standard in the XDC constraints file, and each differential pair is externally supported with a 220 Ω pull-up resistor network to ensure proper voltage swing and signal integrity at high link speeds.

Both FPGAs simultaneously transmit and receive video and audio over separate differential pairs, which require all PMOD B pins. In Section III, we describe these PMOD B pin assignments in detail. To wire the 4 data pins needed for the microphone ADC, we connected them to the servo pins on the board, which we remap via the XDC file.

II. Video Capture and Downsampling

The OV5640 camera module produces 720p RGB data at 30 FPS. We reconstruct the incoming pixel stream using the Lab 5 low-resolution capture pipeline through the `pixel_reconstruct` module, which generates synchronized `h_count`, `v_count`, and valid pixel outputs.

Since the full-resolution stream cannot be transmitted directly due to bandwidth limitations described in Section III, we perform anti-aliased downsampling using a multi-stage pipeline. Our `downsampling_compression` module implements a two-stage anti-aliasing filter: the first stage applies a 3×3 Gaussian blur to the full 1280×720 input, followed by 2× downsampling to 640×360. The second stage applies another 3×3 Gaussian blur to the intermediate resolution, followed by a final 2× downsampling to produce the 320×180 output. This approach significantly reduces aliasing artifacts compared to simple pixel dropping.

We maintain five BRAM-based buffers on each FPGA:

*These authors contributed equally to this work.

- **tx_frame_buffer**: The 320×180 frame that will be transmitted to the remote FPGA uses 57,600 pixels $\times$ 16 bits = 921,600 bits = 115.2 KB.
- **selfie_cam_frame_buffer**: A separately downsampled 160×90 frame displayed locally as a selfie inset uses $8\times$ downsampling (capturing every 8th pixel) to fit within BRAM constraints. This requires only 14,400 pixels $\times$ 16 bits = 230,400 bits = 28.8 KB. The write end captures pixels independently from the main camera stream, while the read end is clocked by `clk_pixel` for display composition.
- **rx_frame_buffer**: This buffer stores packets received from the remote board. Its size is the same as `tx_frame_buffer` (115.2 KB), and its read end is clocked by `clk_pixel` for display composition.
- **tx_audio_buffer**: Ring buffer for audio samples to be transmitted, consisting of 500 samples $\times$ 8 bits = 0.5 KB.
- **rx_audio_buffer**: Ring buffer for received audio samples, which has the same size as `tx_audio_buffer` (0.5 KB).

We refer to these buffers in later sections and describe our total BRAM utilization in Section VIII-A.

III. Physical Link and Differential Signaling

The PMOD B header has 8 pins total, which we allocate as four pins for transmission and four pins for reception on each FPGA. Each direction uses differential signaling with positive and negative pairs, yielding two signals per direction: one for data and one for clock. Specifically, the each FPGA transmits `tx_bit` (differential data) and `tx_dclk` (differential clock), while the receive path uses `rx_bit` (differential data) and `rx_dclk` (differential clock).

We generate the differential clock signal (`tx_dclk`) by counting on the built-in 100 MHz system clock and toggling the output every N cycles, where N is chosen to achieve the target frequency. This is not a true clock signal, but rather a data signal that toggles at a regular rate. We divide the 100 MHz system clock by 5 to produce a 20 MHz differential clock signal. This approach is necessary because the PMOD B pins are not clock-capable pins on the Spartan-7 FPGA, so we cannot use clock generation IP blocks or route actual clock signals to these pins. Instead, we treat the differential clock as a data signal that the receiver samples and uses to determine when to sample the data line. The receiver uses edge detection on this toggling signal to reconstruct the sampling clock.

We set all PMOD B signals designated for differential use to `TMDS_33`, enabling the FPGA's internal termination and suitable I/O behavior for high-speed signaling. We implement the differential pairs using `OBUFDS` and `IBUFDS` primitives for transmission and reception, respectively.

A critical design challenge we encountered was metastability when sampling the differential clock (`rx_dclk`) and data (`rx_bit`) signals from the `IBUFDS` primitives. The `IBUFDS` outputs are asynchronous to our 100 MHz system clock domain, and directly sampling these signals can lead to metastability issues that cause missing clock edges and packet desynchronization. We resolved this by implementing a two-stage synchronizer: the unstable outputs (`rx_dclk_unstable` and `rx_bit_unstable`) from `IBUFDS` are passed through two flip-flops clocked by `clk_100_passthrough` before inputted to the packet deserializer. This synchronization chain ensures that metastable values have time to resolve before being used, dramatically improving link reliability and eliminating the clock missing issues we observed during initial testing.

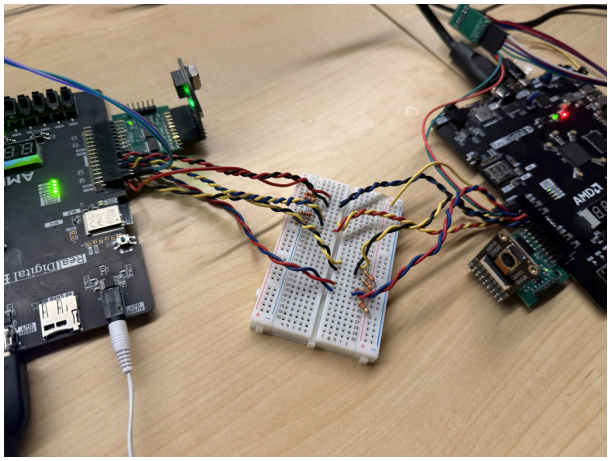


Fig. 2: Complete wiring setup showing both Urbana FPGAs connected via PMOD B cables with external pull-up resistor networks, cameras, HDMI outputs, and microphone connections.

As in Figure 2, we add external pull-up resistor networks to bias each differential pair into a stable region, reducing jitter and promoting clean edges. We tested multiple resistance values through several iterations, optimizing for signal smoothness while ensuring adequate peak-to-peak voltage swing. After evaluation, we settled on 220 Ω pull-up networks, which provide reliable operation at our 20 MHz transmission rate.

IV. Packet Format and Bandwidth Requirements

We structure each transmitted unit as an 18-bit packet:

[2-bit header] [16-bit payload]

We encode the headers as follows:

- **11**: Pixel data (16-bit RGB565 pixel in payload)
- **01**: Frame-start marker
- **10**: Sync packet header
- **00**: Audio data (8-bit audio sample in bits [15:8], bits [7:0] are zero)

Given a frame of $320 \times 180 = 57{,}600$ pixels, the absolute minimum link bandwidth required for our initial goal of 10 FPS would be the following:

$$57{,}600 \times 18 \text{ bits} = 1.0368 \text{ Mbits/frame}$$

$$10 \text{ frames/s} \Rightarrow 10.368 \text{ Mbit/s}$$

With overhead for frame-start markers (100 packets), periodic sync packets (one every 20 pixels for a total of 2,880 packets), and interleaved audio packets (also one for every 20 pixels for a total of 2,880 packets), we transmit $100+57{,}600+2{,}880+2{,}880 = 63{,}460$ packets per frame. At 18 bits per packet, each frame requires $63{,}460 \times 18 = 1{,}142{,}280$ bits. Operating at 20 MHz provides 20 Mbit/s, yielding an effective frame rate of approximately 17.5 FPS, well exceeding our original 10 FPS target.

V. Transmit and Receive Pipelines

We organized the `tx_video` transmission module as a five-state FSM:

1) **SYNC**: On start or reset, send repeated sync packets for initial link alignment. We describe this in more detail later in this section.
2) **FRAME_START**: Emit frame-start markers to indicate a new frame.
3) **PIXELS**: Stream 57,600 pixel packets directly from `tx_frame_buffer`. After every 20 pixels, we transition to `ALIGN_AFTER_PIXEL` to send periodic resync packets.
4) **ALIGN_AFTER_PIXEL**: Send a sync packet to allow the receiver to realign if packet boundaries have drifted. This periodic resynchronization prevents accumulation of bit errors over long frame transmissions. After a single sync packet, we transition to `AUDIO` to send one audio packet.
5) **AUDIO**: Send one audio packet from the `tx_audio_buffer` before returning to `PIXELS`. Since we interleave audio with video by sending one audio packet per every 20 pixels, we maintain synchronized playback.

The interface of `tx_video` is designed to be modular: it outputs a lookup index (`lookup_pixel_index`) into the `tx_frame_buffer` BRAM once the packet serializer asserts `ready_for_next_packet`. The serializer maintains a queue of length 2, ensuring that data is always available for transmission without waiting for BRAM read latency. This buffering allows continuous packet streaming even when BRAM access takes multiple clock cycles.

We deliver each packet to the `packet_serializer`, which shifts bits MSB-first onto the differential data line on each rising edge of the transmitted differential clock. The serializer operates in the 100 MHz domain and synchronizes the `tx_clk` signal to prevent metastability. We manage backpressure via the serializer's `ready_for_next_packet` signal.

A. Synchronization Behavior and *SYNC* State Operation

To achieve reliable frame decoding, both FPGAs must agree on the 18-bit packet boundaries. Because the PMOD link provides only a forwarded differential clock signal (which is actually a toggling data signal) and a single data bit, the receiver has no inherent knowledge of where packet boundaries begin. In standard SPI, the CS line provides this framing, but we lack spare PMOD pins to include it. We therefore resolve this by implementing a syncword-based alignment system. The syncword is the reserved 18-bit pattern:

`FULL_PACKET_SYNC = 18'b101100110101110010`

We chose this pattern to have low autocorrelation, ensuring that it is unlikely to appear in normal pixel traffic. The syncword serves as a unique marker that allows the receiver to identify packet boundaries regardless of the current bit alignment state.

1) **Serializer behavior during SYNC:** During the SYNC state, `tx_video` repeatedly asserts `next_packet_valid` with the sync pattern. The packet serializer then shifts the 18-bit word MSB-first on each rising edge of the differential `tx_clk`. Since the SYNC pattern is repeated thousands of times, the receiver is guaranteed to observe this 18-bit sequence aligned to bit boundaries at some point, even if initially misaligned due to startup conditions or metastability.
2) **Deserializer alignment search:** The packet deserializer runs a simple two-state FSM, where the receiver samples the incoming data on the falling edge of the forwarded differential clock to maximize timing margins across the link:
   - *UNALIGNED*: On every `rx_clk` falling edge, the incoming bit is shifted into an 18-bit buffer. After each shift, the buffer contents are compared against `FULL_PACKET_SYNC`. Since SYNC packets arrive continuously, every possible bit phase is tested. When the buffer exactly matches the sync pattern, alignment is declared.
   - *ALIGNED*: Once aligned, the deserializer increments a bit counter with each `rx_clk` falling edge and outputs one valid packet every 18 bits. Critically, on every clock edge, the deserializer also checks whether the last 18 bits (the current shift register contents) match the sync pattern. If a sync pattern is detected, the bit counter is immediately reset to zero, realigning packet boundaries. This self-correcting behavior ensures that if the system becomes misaligned due to clock drift or bit errors, it automatically recovers when the next periodic sync packet arrives.
3) **Transition from SYNC to FRAME_START:** While the deserializer is aligning, the receive FSM stays in `ST_SEARCH_SYNC`. When the first valid SYNC packet is emitted, the FSM transitions to `ST_WAIT_FRAME_START`. Only after a correct frame-start header (`01`) is received does the receiver advance to the pixel-capture state.
4) **Self-correcting periodic resynchronization:** The system implements a self-correcting alignment mechanism that makes it tolerant to clock skew and transient bit errors. The transmitter sends a syncword packet after every 20 pixel packets (via the `ALIGN_AFTER_PIXEL`

state). In the deserializer's ALIGNED state, on every clock edge, the last 18 bits are checked against the sync pattern. If a match is detected, the bit counter is immediately reset to zero, realigning packet boundaries regardless of the current bit position. This ensures that if misalignment occurs due to clock drift or bit errors, the system automatically recovers when the next periodic sync packet arrives, without requiring a full resynchronization sequence or system reset.

5) **Stability across frames:** After initial alignment, packet boundaries are maintained across all subsequent packets through the periodic resynchronization mechanism. The continuous syncword detection in the ALIGNED state provides automatic correction of any drift, making the system robust to clock domain differences and signal integrity issues. The initial SYNC state is only re-entered if the system is reset.

To receive the differential data, we designed a similar path to mirror the transmitter. The packet_deserializer captures bits on rx_clk edges, performs sync-pattern alignment, and emits completed packets to the rx_video FSM.

The rx_video module exposes a similar modular interface to the transmitter: it outputs a write index (fb_write_index), pixel data (fb_write_pixel), and a write enable signal (fb_write_en) for writing into the rx_frame_buffer BRAM. This interface allows the receive FSM to operate independently of the BRAM implementation details.

The rx_video module transitions through the following states:

1) **SEARCH_SYNC**: Detect the repeated sync pattern from the deserializer.
2) **WAIT_FRAME_START**: Wait for the frame-start header (01) after detecting sync.
3) **RECEIVE_DATA**: Process incoming packets. When a packet with header 11 (pixel data) is received, write it sequentially into rx_frame_buffer. When a packet with header 00 (audio data) is received, write the audio sample into rx_audio_buffer. Sync packets detected during this state trigger realignment in the deserializer. When all 57,600 pixels have been received, transition back to WAIT_FRAME_START.

VI. Display Composition

We use the existing lab pipeline for HDMI rendering. The display composition logic reads from rx_frame_buffer and selfie_cam_frame_buffer, then generates the final composite output. The remote video is displayed at 4× zoom (320×180 scaled to fill 1280×720) as the background. The selfie view is positioned in the bottom-right corner with a 20-pixel margin from the screen edges, displayed at 2× zoom (160×90 scaled to 320×180), as shown in Figure 3. We render a 5-pixel-wide pink border (RGB565 color 0x1ea0) around the selfie view to visually distinguish it from the remote video feed.



Fig. 3: Display composition showing remote video with selfie view overlay and green border. The remote video (320×180) is displayed at 4× zoom filling the 1280×720 screen, while the selfie view (160×90) is displayed at 2× zoom in the bottom-right corner.

The system includes a zooming feature: pressing button 3 toggles between the normal and "bigview" modes. In bigview mode, the selfie camera is displayed at 4× zoom and the remote video is darkened to 25% brightness.

VII. Audio Integration

Audio functionality is fully integrated with the video transmission and reception pipelines by interleaving the audio packets between video packets. We implement ring buffer interfaces using standard dual-port BRAM primitives. While we logically treat these as ring buffers with separate read and write pointers, they are physically implemented as regular BRAM primitives with address wraparound handled in the control logic.

The audio BRAM size is calculated based on the target audio specifications: 8 ksps sample rate, 8-bit samples, and a buffer depth of 500 samples. The size is somewhat arbitrary, as because of the ring buffer logic we any size buffer works as long as it is greater than the amount of samples in $1/17.5$ seconds ($8000/17.5 = 457$) This yields as follows:

$$500 \text{ samples} \times 8 \text{ bits} = 500 \times 8 \text{ bits} = 4000 \text{ bits per buffer.}$$

To capture audio, we implemented the Lab 2 microphone ADC interface, which reads audio samples from the microphone over SPI at 8 ksps. The write side of the tx_audio_buffer ring buffer stores incoming samples from the ADC, while the read side supplies samples to the tx_video state machine for packetization. Audio packets use header 00 and contain the 8 bit audio sample in bits [15:8] of the payload, with the remaining 8 bits as zeros.

On the receiving FPGA, the rx_video state machine detects audio packets by their header 00 and writes the audio samples into the rx_audio_buffer ring buffer. A separate read counter, synchronized to the 8 kHz sample rate, reads samples from this buffer and outputs them via PWM to the headphone jack. The ring buffer structure allows for continuous audio streaming even when video packets are being processed.

Audio packets are interleaved with video packets: after every 20 pixel packets, the transmitter sends one audio packet before continuing with more pixels. This interleaving ensures that audio and video remain synchronized while maintaining a high video frame rate. To accommodate the ADC wiring requirements, we connected the four data pins needed for the microphone's SPI interface to the servo pins on the board, which we repurposed for GPIO use via the XDC constraints file. This repurposing was necessary because the camera occupies the entire PMOD A header, leaving only PMOD B available for the video link.

VIII. Resource Utilization and Timing

A. BRAM Utilization

Each FPGA maintains five dual-port BRAMs for video and audio buffering:

- **tx_frame_buffer**: 57,600 entries × 16 bits = 921,600 bits = 115.2 KB
- **rx_frame_buffer**: 57,600 entries × 16 bits = 921,600 bits = 115.2 KB
- **selfie_cam_frame_buffer**: 14,400 entries × 16 bits = 230,400 bits = 28.8 KB
- **tx_audio_buffer**: 500 entries × 8 bits = 4,000 bits = 0.5 KB
- **rx_audio_buffer**: 500 entries × 8 bits = 4,000 bits = 0.5 KB

Thus the total BRAM utilization per FPGA is 260.2 KB, well within the available resources of the Spartan-7 FPGA on the Urbana board. A primary design constraint throughout the project was minimizing BRAM usage. To that end. the selfie camera buffer is stored at an 8× downsampled resolution, specifically to reduce BRAM usage while still providing a visible selfie overlay. Since the selfie view only occupies a small on-screen region, this aggressive downsampling is visually acceptable.

Additional BRAM usage could be saved by reducing the audio buffer depth. Although we calculated that storing about 400 samples (50 ms of audio) is sufficient, we choose a conservative size of 500 samples for better continuity margins. If tighter BRAM constraints were present, we can reduce audio storage with minimal impact on system performance.

B. Timing Requirements

The system operates across multiple clock domains:

- **Camera domain** (`clk_camera`): 200 MHz for sampling camera data
- **System domain** (`clk_100_passthrough`): 100 MHz for packet processing
- **Display domain** (`clk_pixel`): 74.25 MHz for HDMI output
- **Differential clock signal** (`tx_dclk`/`rx_dclk`): 20 MHz toggling signal (not a true clock) for link transmission

The two-stage synchronizers on the differential receive paths ensure proper clock domain crossing from the asynchronous `IBUFDS` outputs to the 100 MHz system clock. All BRAMs use dual-port configurations with independent clocks for read and write ports, allowing asynchronous operation between camera capture, packet processing, and display rendering.

After queueing a packet for transmission, it must wait 18 clock cycles to finish transmitting. On the receiving end, it immediately starts building the packet. Since we designed the system to transmit and receive packets continuously without delay, each packet takes about 1 µs to be transmitted from one FPGA and get deserialized by the other FPGA. We have syncword and framestart packets as overhead, as explained in Section IV, to make the system work. Together, this gives approximately 17.5 FPS of video, which appears continuous and exceeds our goal of 10 FPS.

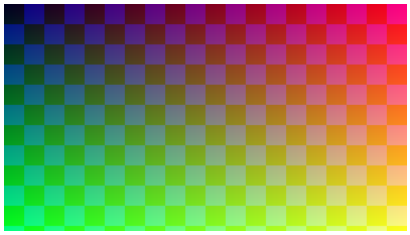


Fig. 4: Image we transmitted in simulation.

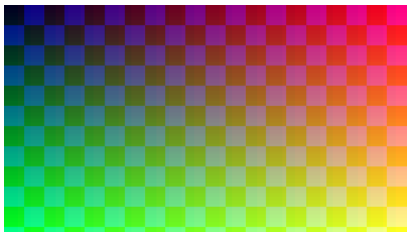


Fig. 5: Simulated reconstruction result. The received image is visually identical to the source image and verified pixel-accurate in code, with minor artifacts from color space conversion.

IX. Evaluation

A. Simulation

We created cocotb testbenches and examined their waveforms for smaller modules. In particular, to verify the pipeline for both `tx_video` serialization and `rx_video` deserialization, one test instantiates both modules connected by virtual wires (omitting the differential signaling for simplicity in simulation as it just passes through the IPs). The modular interface design allows us to swap the BRAM implementations for Python arrays in the testbench. Specifically, we replace the `tx_frame_buffer` BRAM with a Python array containing the source image, and the `rx_frame_buffer` BRAM with another Python array for the received image.

The testbench leverages the exposed interfaces: `tx_video` outputs `lookup_pixel_index` which we use to read from the Python array, and `rx_video` outputs `fb_write_index`, `fb_write_pixel`, and

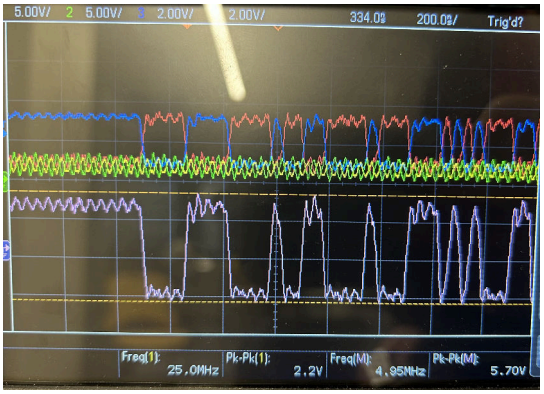


Fig. 6: Measured differential waveforms at 25 MHz test frequency. The yellow and green traces are the differential clock positive and negative lines, respectively. The blue and red traces are the differential data positive and negative lines, respectively. The purple trace shows the result of subtracting the differential data lines (DATA_P − DATA_N), clearly displaying the digital signal transitions. Signal quality at this frequency validated our design, though we operate at 20 MHz for our final implementation.

`fb_write_en`, which we use to write into the receive Python array. After transmission completes, we write the received array to an image file for visual comparison.

One image we simulated transmitting is in Figure 4, and its reconstruction is in Figure 5. The reconstructed images are visually identical to the source images, with the exception of artifacts visible when zooming in. These artifacts arise from the conversion between the full color space in the source images and the RGB565 format we use in the framebuffers. We verified pixel-level accuracy in code by comparing the transmitted and received pixel arrays, confirming perfect reconstruction of the RGB565 data.

B. Oscilloscope Testing

During development, we tested the differential link at multiple frequencies to characterize signal integrity and determine the optimal operating point. We probed the differential clock and data lines at 5 MHz, 12.5 MHz, and 25 MHz, configuring the oscilloscope in subtraction mode to cleanly resolve the differential pairs. Figure 6 shows the measured waveforms at 25 MHz, demonstrating that signal quality remains sufficient for reliable packet decoding even at this higher frequency. The differential clock signal (`tx_dclk`) appears as a regular toggling signal, while the data line shows the serialized packet data with clean transitions. Based on these measurements and our bandwidth requirements, we selected 20 MHz as our operating frequency, providing sufficient headroom above the minimum 10.368 Mbit/s requirement while maintaining excellent signal integrity. The two-stage synchronizers on the receive path ensure proper sampling of these signals despite their asynchronous nature relative to the system clock.

C. Live System Demonstration

The complete system has been successfully demonstrated operating bidirectionally between two FPGAs, achieving approximately 18 FPS with 320×180 video resolution and 8 kHz audio. The self-correcting alignment system with periodic syncword transmission ensures robust operation, and the two-stage synchronizers on the receive path eliminate metastability issues. The anti-aliasing pipeline significantly improves visual quality compared to simple pixel dropping downsampling.

X. Conclusion

We have successfully implemented and demonstrated FPGAceTime, a bidirectional audio/video communication system between two FPGAs. The system features anti-aliased video downsampling, integrated audio capture and playback, robust packet-based communication over differential signaling with self-correcting alignment, and display composition with selfie overlay and zooming capabilities.

Key design insights include: (1) proper clock domain crossing with two-stage synchronizers to handle metastability from IBUFDS primitives, which solved critical clock missing issues; (2) the self-correcting syncword-based alignment system that continuously monitors for packet boundaries and automatically realigns; (3) treating the differential clock as a toggling data signal rather than a true clock, necessary because PMOD pins are not clock-capable; and (4) modular interfaces that enable independent development and testing.

The system demonstrates reliable bidirectional operation at approximately 18 FPS with 320×180 video and 8 kHz audio, operating at 20 MHz link speed, exceeding our original 10 FPS goal.

A. Future Work

While we have successfully met and exceeded our original goal of 10 FPS with full audio integration, several enhancements remain. More rigorous handling of metastability and clock domain crossings throughout the design would further improve robustness. Compression via color tables or palette-based schemes could improve video quality at similar bit rates. With our periodic syncwords, we ended up implemented a self-correcting mechanism for packet misalignment, meeting our stretch goal for generic error correction. However, packet-wise error correction coding (CRC or forward error correction) would further improve reliability in noisy environments.

Overall, we are satisfied with the system we have achieved. Exceeding our original 10 FPS target and successfully integrating full bidirectional audio represents significant progress. The self-correcting alignment system, anti-aliasing pipeline, and comprehensive feature set demonstrate a robust and functional video conferencing system.

B. Code Repository and Demo Video

https://github.mit.edu/6205F25/fa25-6205-team46
https://youtu.be/gpY-w5pYmGA?si=eMdQjoGhFoKZxJhA

C. Author Contributions

Both Annie and Sanjith contributed equally on developing modules for the project. Both authors worked on the transmission and reception state machines. Annie worked on the wiring for the differential signaling and identifying the correct pull-up resistance via testing on the scope. Sanjith wrote the packet serialization and deserialization modules.

Both authors contributed equally to the writing of this report. Annie mainly wrote Sections II, III, and VII. Sanjith mainly worked on Sections IV, V, and VI. The authors wrote the other sections together.

D. Acknowledgments

This project would not have been possible without the support of Professor Joe Steinmeyer, the 6.205 course staff, and especially our assigned TA Yoland Gao. Both Yoland and Joe were very helpful in debugging our video transmission issues. We are very grateful for their support and inspiration that have helped us achieve more ambitious goals.

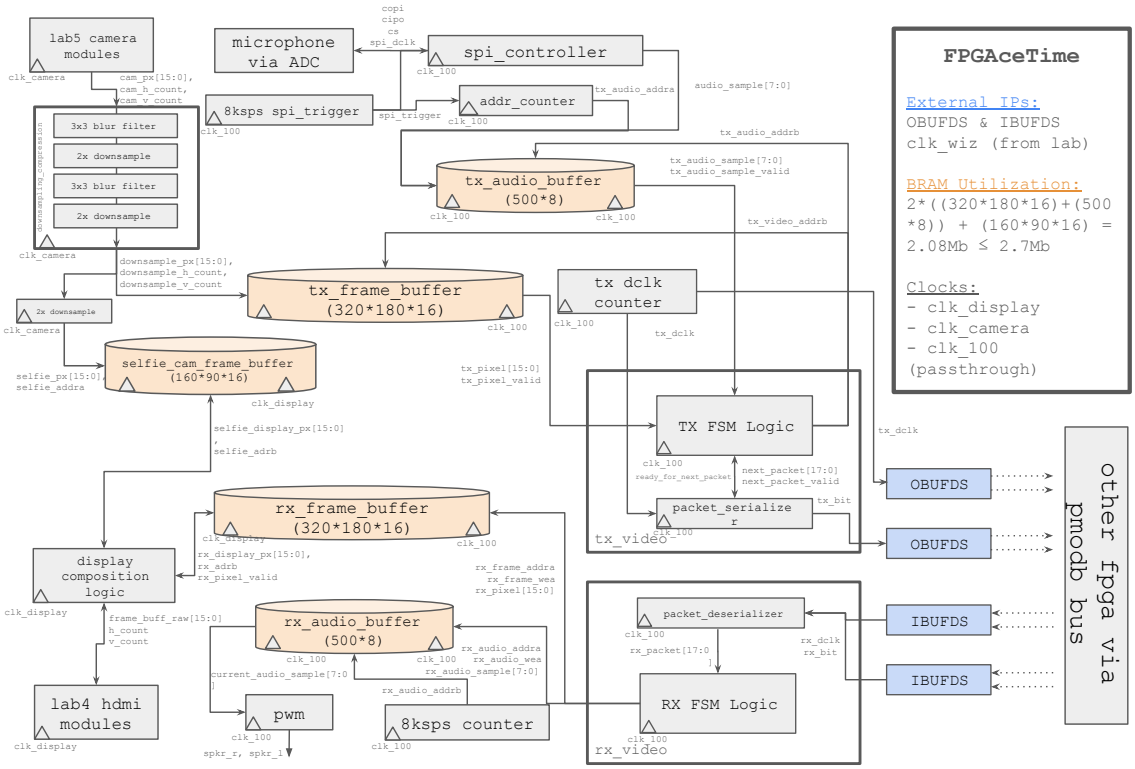


Fig. 7: System block diagram showing the complete bidirectional audio/video data flow. Both FPGAs operate identically, with simultaneous transmit and receive pipelines.