FPGA-Time

Peyton Shields & Jade Sund
{pshields, jcsund}@mit.edu

Introduction

FPGA-Time is a communication system which enables two users to communicate with audio and video. It uses a camera and microphone to record one user, composes the data into a packet, forwards it over ethernet to another FPGA, then reconstructs the video feed on a VGA display and outputs audio through an external speaker.

To chat with another user, each FPGA has the ability to “call” someone connected to it over Ethernet . Using the buttons on the board, you are able to initiate, accept, or deny a call. Communication begins after the called user accepts the request to chat. To accept the request, the called user will see an incoming call notification on their VGA display. They then are able to accept or decline the call using buttons on the FPGA.

During a video call, there are optional effects you can apply to the audio and video (similar to FaceTime). This includes a random noise effect, an overlay image (e.g. a hat), and more. Each user can also mute their microphone and toggle their camera on and off.

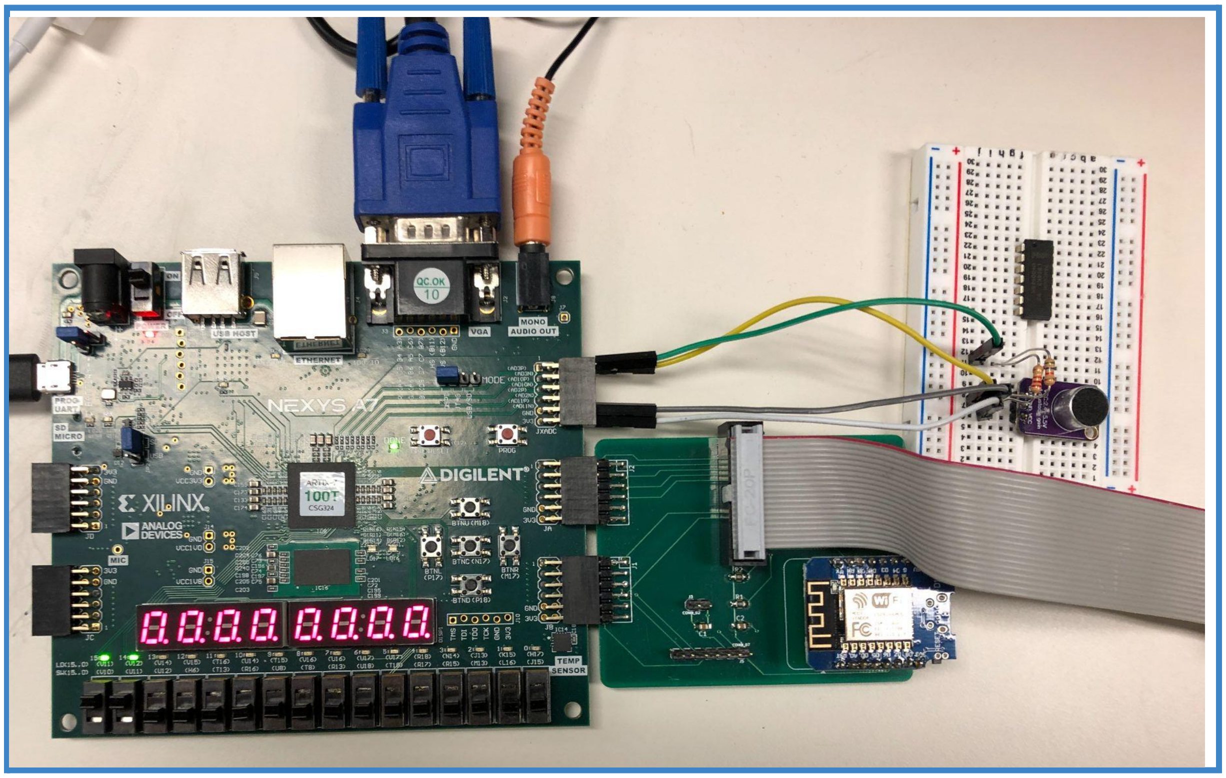
Overview

The system is divided into four primary submodules. The camera controller handles capturing video data from an external camera and packaging the pixel data into a video payload. The audio controller similarly captures and packages external microphone data into an audio payload, but also plays back received audio data.

The ethernet controller takes the built audio and video payloads, packages them together with metadata for state transitions and audio visual effects, then forwards the packet via UDP over a wired ethernet connection. Upon receipt of a packet, the ethernet controller decapsulates packets into their containing audio, video, and metadata, performing any necessary clock-crossing along the way.

To display the received video data and apply visual effects, the display controller ensures synchronization between sent versus received video as well as draws pixels to the screen. Its primary purpose is to provide an abstraction from the camera packet creation logic, enabling easy customization to the user interface. Each controller’s inputs, outputs, and behaviors are shown below.

FPGA Setup



The FPGA connects to a display through VGA

Video chatting with another user

Block Diagrams – Jade - Distributed Throughout Write Up

Camera Controller – Peyton

The camera controller is in charge of reading camera data, outputting a pixel for your own video (to see yourself during a call), and creating video payloads for use by the ethernet controller. *The camera controller accepts hcount and count signals from an XVGA module running on a 65 MHz clock with a resolution of 640x480 pixels.

The camera controller interfaces with an external camera which is itself controlled by an ESP8266 microcontroller. Camera data is read on a 65 MHz clock at a resolution of 320x240 pixels which each are in 12 bit color. The camera read module was designed by Joe Steinmeyer.

When reading pixels from the camera, information is stored in a dual port frame buffer. Pixel data is written to the BRAM on the camera's clock and extracted at 65 MHz. The extracted pixel data can then be directly drawn on the user's screen. *This is the case for displaying your own video feed during a video call.

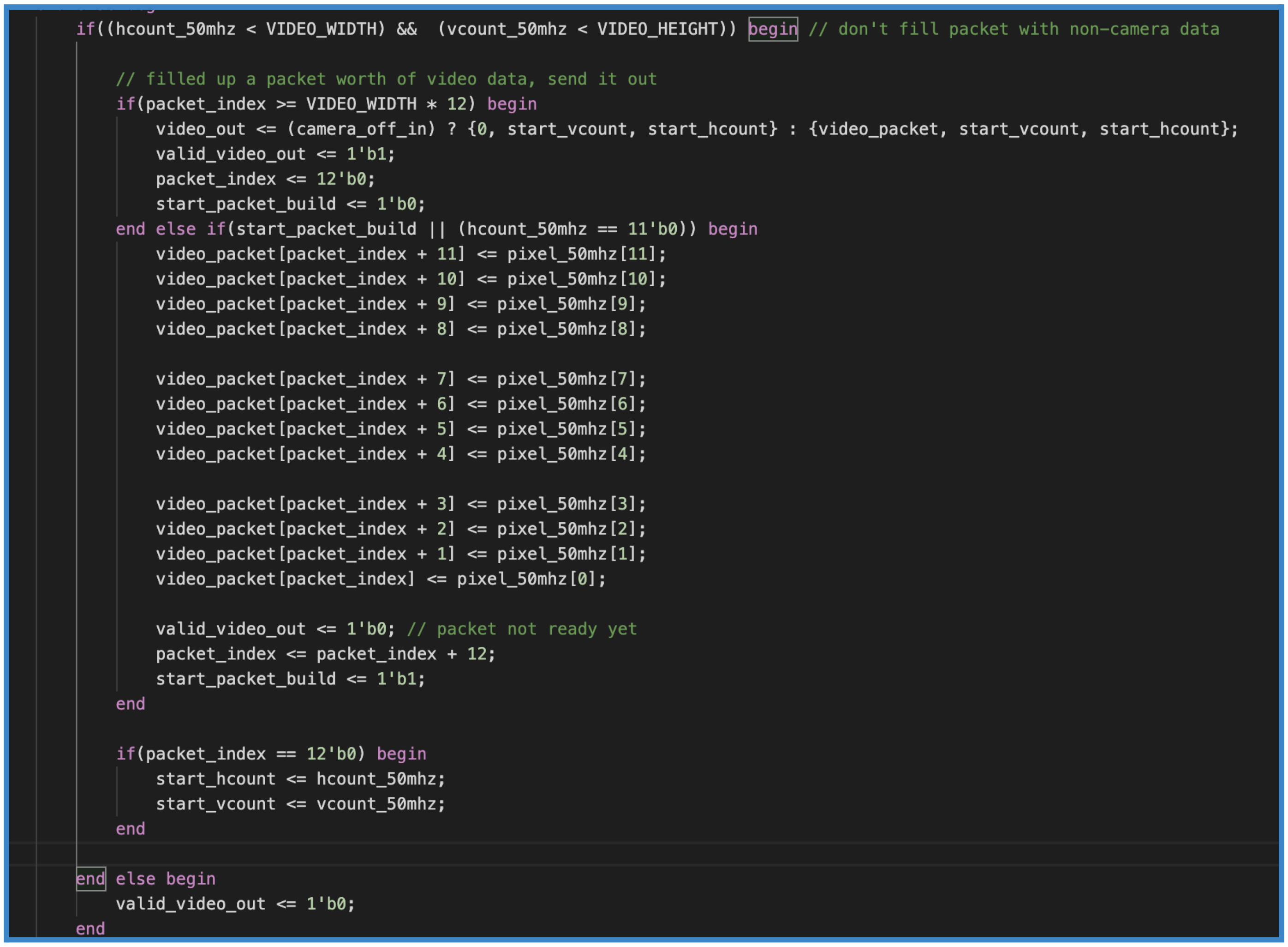
To create the video payload, we first cross the pixel data (and hcount/vcount) from the 65 MHz clock domain into the 50 MHz domain. This allows us to output a video payload at the same clock rate used by the ethernet controller. To perform this clock crossing, we use another dual port BRAM with a write width of 33 bits. We input the current pixel as well as hcount and vcount for that pixel to the BRAM, then extract the same data on a slower 50 MHz clock.

After performing the clock crossing, we begin creating the video payload. The video payload is 483 bytes, which is enough to send an entire line (320) of 12 bit pixels. The video payload also contains the starting hcount and vcount for the line of pixels contained within the payload. *Because we send the entire line of pixels, the start hcount is always equal to 0.

To build the payload, we create a 12 bit payload index which starts at 0 as well as a 483 byte logic. When the packet index is 0, the system waits until hcount is also 0 to begin payload construction. This ensures that we send the entire 320 pixels on one line. For each new pixel in the line, we store the 12 pixel bits at the payload index, increment the payload index by 12 (for each bit in the pixel), and repeat the process until 320 pixels are contained within the video payload. At this point, we concatenate the starting hcount and vcount for the line of pixels and output this value for use by the ethernet controller.

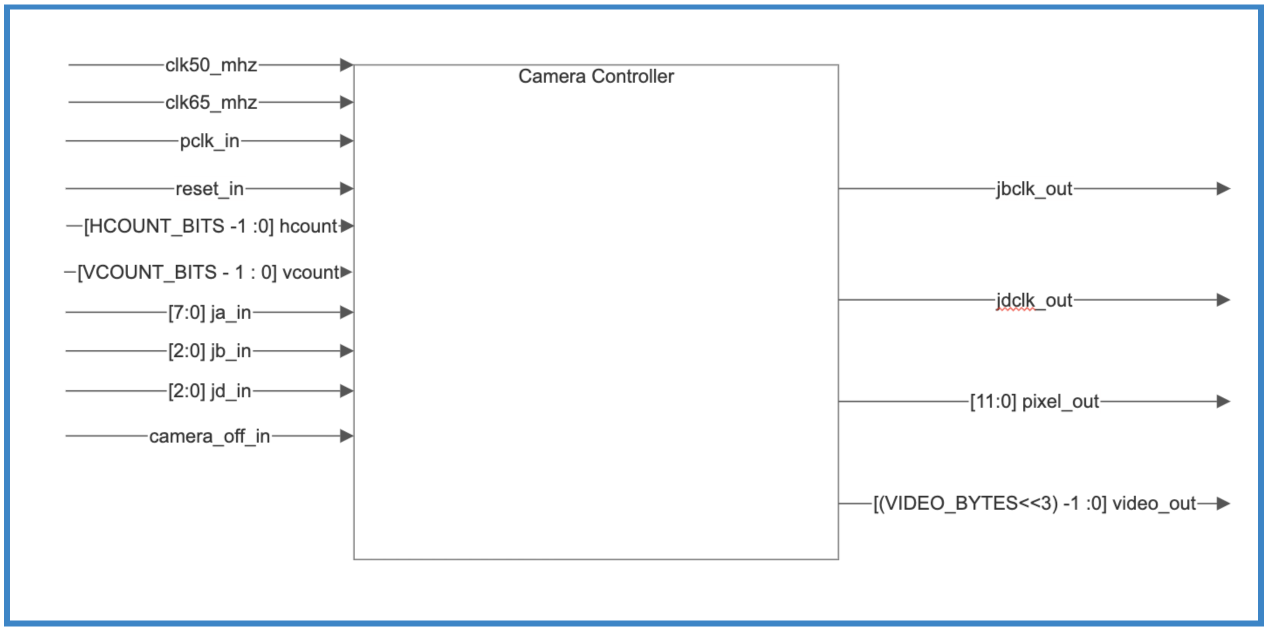
One of the biggest challenges with this module was deciding how many pixels to send at a time and how to ensure synchronization when redrawing on the receiver's display. Our initial approach was to include the start hcount and vcount for the first pixel in the payload, but not enforce that the start hcount must be 0. This equates to sending 320 pixels, but with possible wrapping from one line to the next. In practice, this worked but yielded jagged edges on received video and overall poor quality. We also experimented with building the video payload on a 65 MHz clock then using a BRAM to shift the 483 byte payload to the 50 MHz domain, but this proved unfruitful. In our testing, no video would display with this approach.

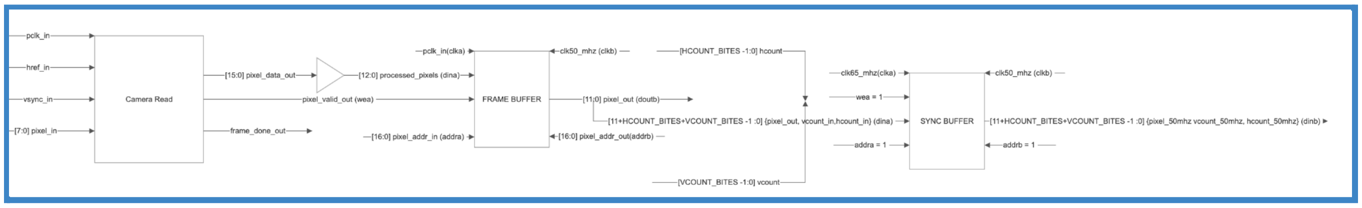
If we were to write this module from scratch, we would spend more time ensuring that a video payload contains every pixel in a line, in the order they appear on the line. One of the cosmetic flaws in our system is the replication of video on a receiver's screen. After two thirds of the image, the same image is drawn again. We believe that this is due to an error in building the video payload, likely caused by the desynchronization of the hcount, vcount, and pixel data. Pixel data goes through two clock domain crossings and all three must go from a faster to a slower clock. The metastability arising from multiple crossings is likely the culprit of the duplicated video.



```
if((hcount_50mhz < VIDEO_WIDTH) &&  (vcount_50mhz < VIDEO_HEIGHT)) begin // don't fill packet with non-camera data

    // filled up a packet worth of video data, send it out
    if(packet_index >= VIDEO_WIDTH * 12) begin
        video_out <= (camera_off_in) ? {0, start_vcount, start_hcount} : {video_packet, start_vcount, start_hcount};
        valid_video_out <= 1'b1;
        packet_index <= 12'b0;
        start_packet_build <= 1'b0;
    end else if(start_packet_build || (hcount_50mhz == 11'b0)) begin
        video_packet[packet_index + 11] <= pixel_50mhz[11];
        video_packet[packet_index + 10] <= pixel_50mhz[10];
        video_packet[packet_index + 9] <= pixel_50mhz[9];
        video_packet[packet_index + 8] <= pixel_50mhz[8];

        video_packet[packet_index + 7] <= pixel_50mhz[7];
        video_packet[packet_index + 6] <= pixel_50mhz[6];
        video_packet[packet_index + 5] <= pixel_50mhz[5];
        video_packet[packet_index + 4] <= pixel_50mhz[4];

        video_packet[packet_index + 3] <= pixel_50mhz[3];
        video_packet[packet_index + 2] <= pixel_50mhz[2];
        video_packet[packet_index + 1] <= pixel_50mhz[1];
        video_packet[packet_index] <= pixel_50mhz[0];

        valid_video_out <= 1'b0; // packet not ready yet
        packet_index <= packet_index + 12;
        start_packet_build <= 1'b1;
    end

    if(packet_index == 12'b0) begin
        start_hcount <= hcount_50mhz;
        start_vcount <= vcount_50mhz;
    end

end else begin
    valid_video_out <= 1'b0;
end
```

Video payload creation




pclk_in
href_in
vsync_in
[7:0] pixel_in
Camera Read
[15:0] pixel_data_out
pixel_valid_out (wea)
frame_done_out
[12:0] processed_pixels (dina)
pclk_in(clka)
[16:0] pixel_addr_in (addra)
FRAME BUFFER
clk50_mhz (clkb)
[11:0] pixel_out (doutb)
[16:0] pixel_addr_out(addrb)
[11+HCOUNT_BITES+VCOUNT_BITES -1 :0] {pixel_out, vcount_in,hcount_in} (dina)
[HCOUNT_BITES -1:0] hcount
[VCOUNT_BITES -1:0] vcount
clk65_mhz(clka)
wea = 1
addra = 1
SYNC BUFFER
clk50_mhz (clkb)
addrb = 1
[11+HCOUNT_BITES+VCOUNT_BITES -1 :0] {pixel_50mhz vcount_50mhz, hcount_50mhz} (dinb)

See Block Diagrams in HTML for a better view of block diagrams.

Audio Controller - Jade

The audio controller creates an audio payload for use by the ethernet controller using samples from the microphone. The audio controller does not deal with unpacking the incoming audio payload. This controller interfaces only with the external microphone and the ethernet controller and operates on a 50mhz clock.

The general function of the audio controller is to downsample incoming microphone data to 6 kHz, filter the downsampled signal using a 31 tap low pass filter, then build the audio payload using the upper eight bits of the filter's output.

The audio controller also handles the audio effects for muting and random noise. When mute is toggled, the controller builds audio payloads that are entirely 0. Engaging the noise effect adds "random" noise to the received microphone data before filtering. In reality, the random noise is just the unfiltered mic input to the controller.

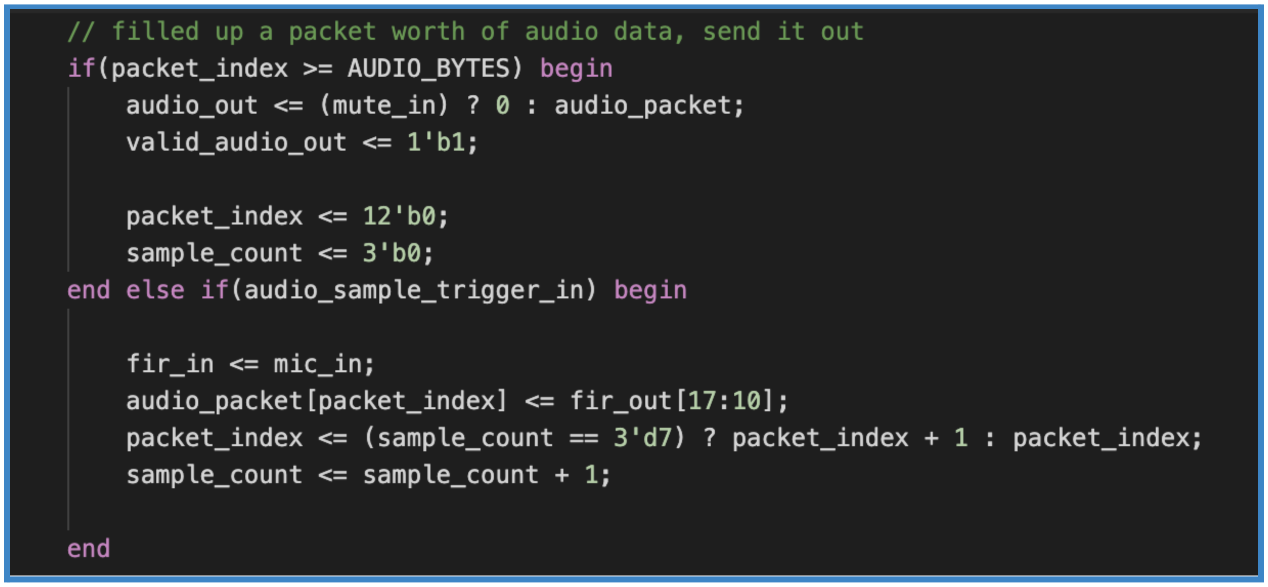
One of the decisions we had to make was how to send audio and how much audio data to send at a time. We played with the idea of alternating audio and video packets, but decided on sending one line of camera pixels and 8 bytes of audio.

Audio playback occurs within top level rather than in the audio controller, yet since it unpacks the payload its operation is tied directly to that of the audio controller.

Audio playback works by playing an alternating tone when the display fsm state is INCOMING, playing the filtered received audio when fsm state is CONNECTED, and playing nothing in all other fsm states. When the display fsm state is CONNECTED, the received audio payload parsed out by the ethernet controller is indexed into using a variable called audio_packet_index to grab eight bits of audio from LSB to MSB(see the bottom of TOP LEVEL in the html file). The eight bits are then passed into the 31 tap fir filter and the most significant eight bits of the output of the fir filter are held on the output for eight cycles to upsample to a 48khz rate.

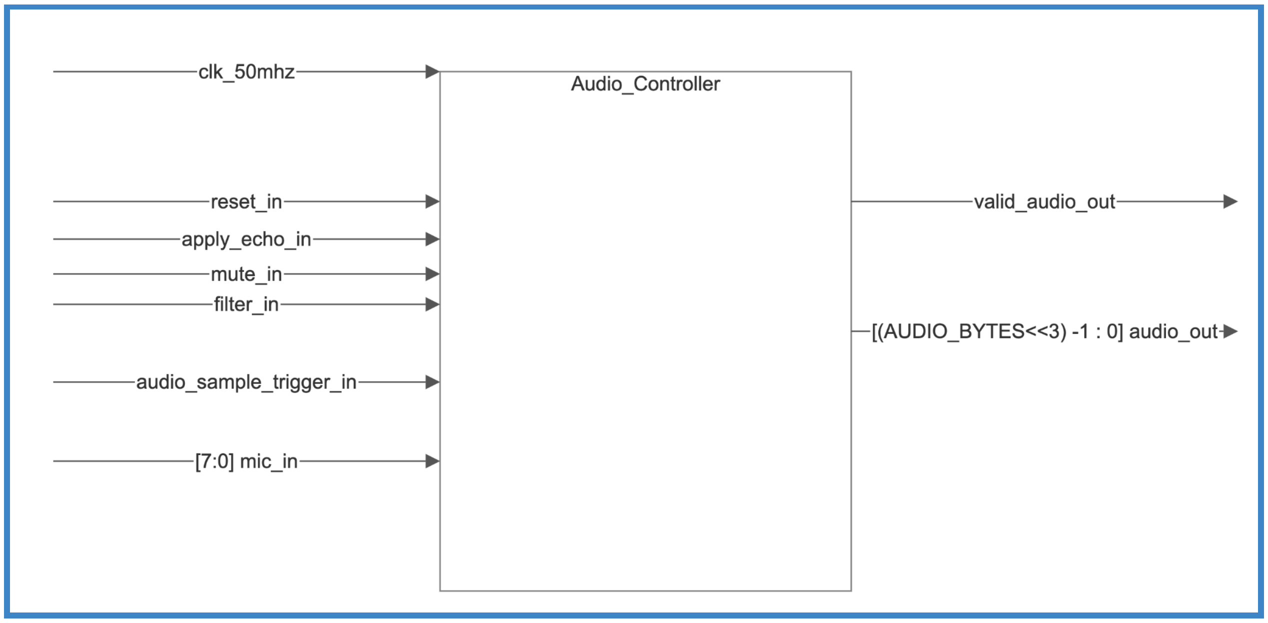
We ran into a few issues while developing the audio controller and the audio playback method. First, the microphone gain was not tuned well enough and caused audio to sound like pure noise even when passed directly from input to output. Once we retrieved a mic with good gain this issue was solved and words were perceptible in audio samples. Second, the raw microphone input contained a lot of high frequency noise. To get rid of this we initially only low pass filtered the input to audio payloads, but there remained a significant amount of high frequency signals. To further weed out high frequencies, we added a low pass filter before outputting the audio signal from a payload. This did get rid of some more high frequency signals, but there remains a significant amount of high frequency information in the audio output.

While trying to get rid of this high frequency information we attempted to implement an audio buffer and use zero holding. We attempted most of these methods before discovering the microphone gain issue and didn't have the time to return to them later, so given more time we would have liked to try different ways to eliminate the remaining high frequency signals such as an audio buffer, zero holding, and increasing the number of taps in our filter.



```
// filled up a packet worth of audio data, send it out
if(packet_index >= AUDIO_BYTES) begin
    audio_out <= (mute_in) ? 0 : audio_packet;
    valid_audio_out <= 1'b1;

    packet_index <= 12'b0;
    sample_count <= 3'b0;
end else if(audio_sample_trigger_in) begin

    fir_in <= mic_in;
    audio_packet[packet_index] <= fir_out[17:10];
    packet_index <= (sample_count == 3'd7) ? packet_index + 1 : packet_index;
    sample_count <= sample_count + 1;

end
```

Audio payload creation



Ethernet Controller – Peyton

The ethernet controller implements send and receive modules for packet transmission via UDP. The send and receive modules were originally created by Babu-Abel, but were modified slightly to support additional packet headers. The send and receive modules operate on a 50 MHz clock, which means that video data (and visual effects metadata) must be safely passed to and from the 65 MHz domain for successful transmission.

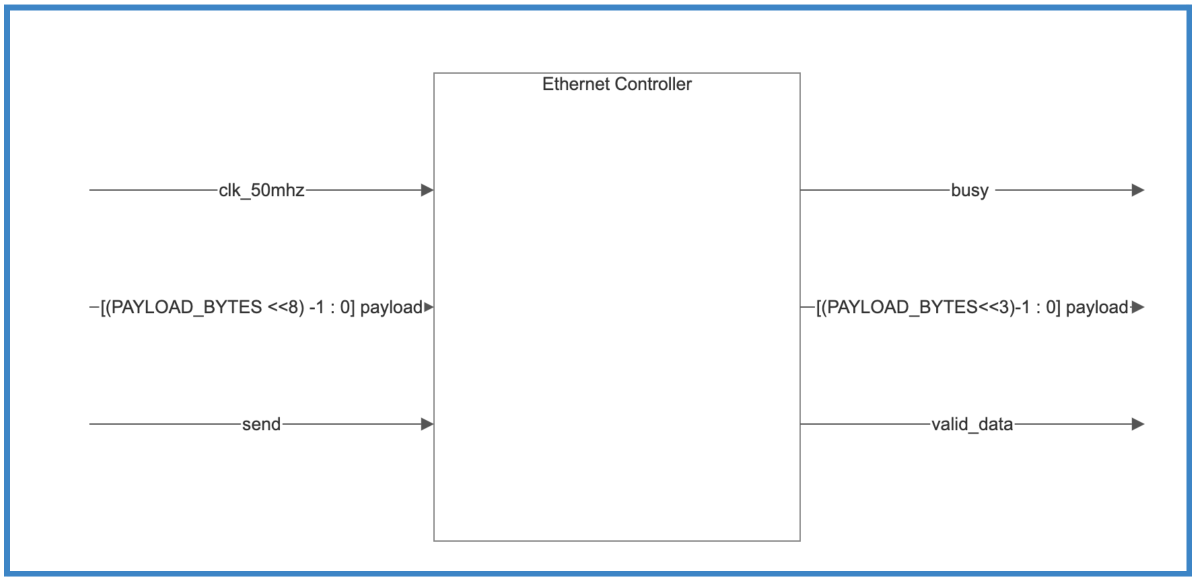
The send and receive modules accept and output, respectively, a 492 byte payload. The maximum ethernet payload size is 1500 bytes, but using a payload this large could cause delays due to packet creation. The payload is made up of 483 bytes of video data, 8 bytes of audio data, 4 bits of state machine transition information (action information), and 4 bits of metadata for effects to apply to the received transmission. We treat the action information and effect metadata as headers which can easily be extracted upon receipt of a packet.

The ethernet controller is also responsible for assembling and parsing payloads. Upon the receipt of a packet, it parses out action information to be sent to the display (call) state machine, video and audio data, and the effects metadata.

Each send and receive module is configured to transmit and listen to packets sent on port 5000 via the private broadcast IP 169.254.255.255. The subnet 169.254 is the automatic private domain used by ethernet and the 255.255 suffix indicates that packets should be treated as if they were directed to everyone. While this is impractical for a setup with more than two users, it requires less configuration for our application.

Additionally, the ethernet controller integrates CRC checks to ensure that transmitted data matches the received data. This helps ensure that no payload bits were changed, producing a better chatting experience.

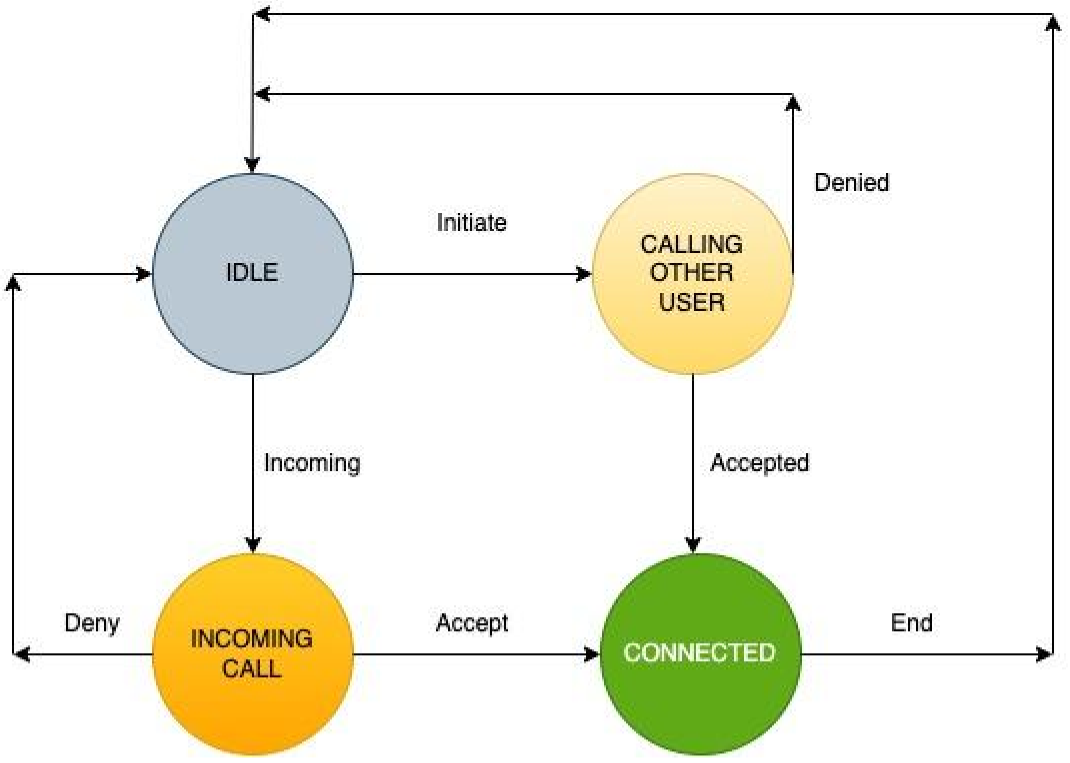
One of the biggest challenges when integrating ethernet is the metastability which arises from differing clock domains. If we were developing the send and receive modules from scratch, matching their clock speed to the rest of our system would be a top priority.

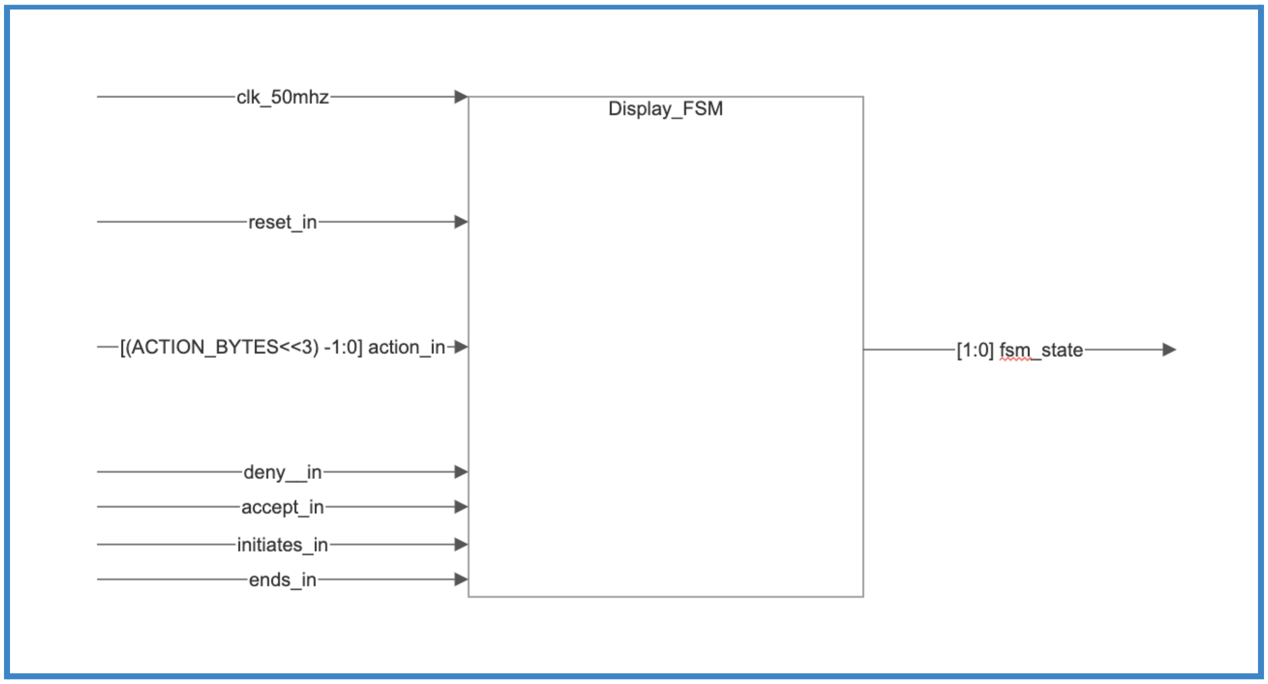

Ethernet Controller
clk_50mhz
[(PAYLOAD_BYTES <<8) -1 : 0] payload
send
busy
[(PAYLOAD_BYTES<<3)-1 : 0] payload
valid_data

Display Controller - Jade

The display controller contains a basic finite state machine. The current state of this state machine determines what visuals need to be printed on the display. The controller runs on a 50mhz clock to avoid synchronization issues with the action_in input that comes from the ethernet. The action_in input can be any of the local parameters IDLE_ACT = 0, CALL_DATA = 1, START_CALL = 2, END_CALL = 3, ACCEPT_CALL = 4, or DENY_CALL = 5. Within the display controller actions are mapped to local variables: accepted = CALL_DATA || ACCPET_CALL, denied = DENY_CALL, incoming = START_CALL, ended = IDLE_ACT. The other inputs deny_in, accpet_in, initates_in, and ends_in are all button inputs from the local FPGA. The potential fsm_states are IDLE, CALLING, INCOMING, and CONNECTED.

The display controller was the only module that we were able to test in simulation, so after testing in simulation the only issue we ran into was having a multi driven port.





Time Domains – Peyton

FPGA-Time operates on 65 MHz and 50 MHz clocks. The ethernet controller must utilize a 50 MHz clock as dictated by the protocol and onboard adapter. We also elected to use a 65 MHz clock for the VGA display as this allows us to draw received video data faster, providing a more seamless experience for the end user.

There are two primary system components which require clock crossing: the effects to apply to received video and the video data to be sent via ethernet.

To cross from the ethernet's 50 MHz clock to the display controller's 65 MHz clock, we use a simple two register shift. In testing, we found that this was enough to resolve any issues with metastability. It should be noted, however, that this approach only works because the display controller runs on a faster clock. This was another design choice to make the development easier.

Crossing clock domains for video data is contrastingly more involved as it must go from the 65 MHz to 50 MHz domain when being packaged and forwarded over ethernet, and from the 50 MHz to 65 MHz domain when a packet containing video data is received.

For the packaging and forwarding crossing, we use a small, dual port BRAM capable of storing 33 bits of information. We store the hcount, vcount, and camera pixel data in the BRAM on a 65 MHz clock and extract the same data on a 50 MHz clock. The read/write latency introduced by the memory helps alleviate metastability issues. The extracted data is then packaged up into a video payload as normal by the camera controller.

To cross clock domains on the receipt of a packet, we utilize another dual port BRAM which acts as a frame buffer. The BRAM writes to a vcount address stored within the received payload on a 65 MHz clock; it writes an entire line (320) of pixels. We then extract the line of pixels based upon the receiver's vcount (from their XVGA module).

While our clock crossing solutions are imperfect, they minimize negative slack and reduce build times by ensuring that the compiler (Vivado) does not attempt to optimize an impossible scenario. We found that any further metastability issues arising from the multiple time domains were imperceptible because of the speed of human perception for audio and video.

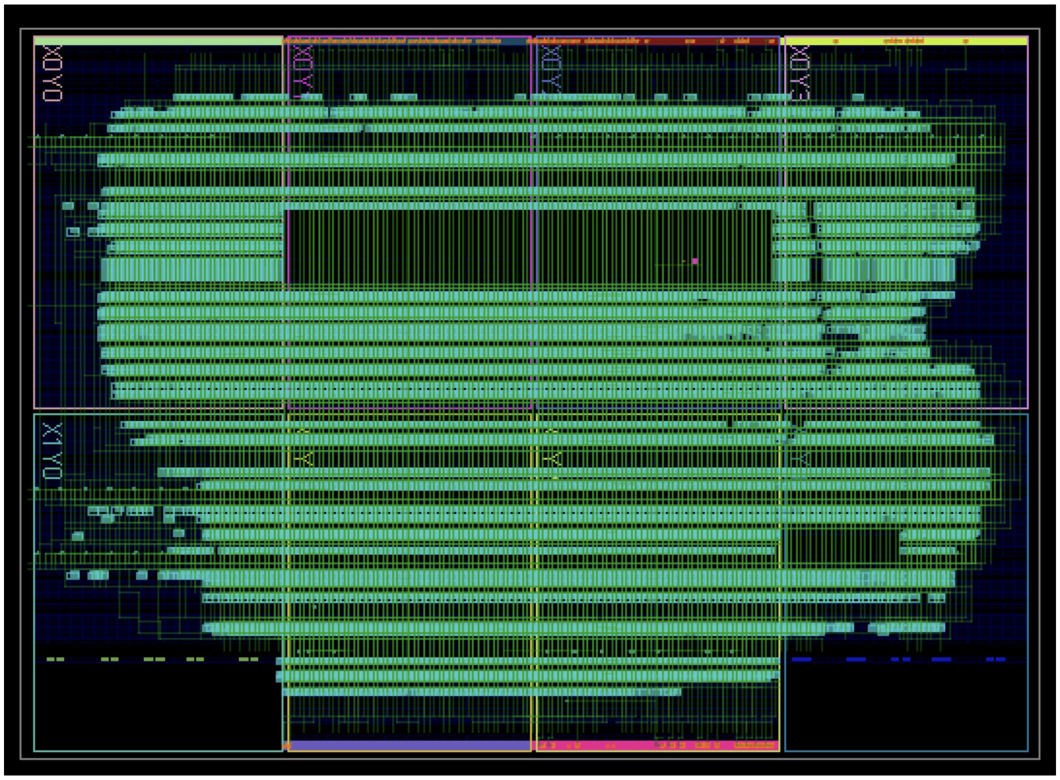
Limitations - Jade

One of the major limitations for this project was BRAM usage. We opted not to use external storage because we didn't want to interface with external memory and we believed that our project was doable with the FPGA memory alone. The final iteration of our project uses 70% of the FPGA BRAM. This is with frame buffers for both the incoming video payloads as well as the user camera data, stored coe files, and various synchronization buffers. Our current implementation only has one stored coe image, a small 32x32 pixel santa hat, so if we wanted to expand the video image overlay we would likely have to interface with external memory or only use black and white pixel information. Furthermore, if we were to expand to having multiple users being engaged in a call, we would run the risk of running out of BRAM space due to needing an additional frame buffer for every additional user.

We also ran into issues with the microphone quality. If the gain on the microphone is not well tuned, the audio output is unrecognizable. Another limitation on audio was the effectiveness of the 31 tap low pass filter. Filtering post downsampling and pre upsampling cleared many of the high frequency signals that made the signal unclear, but we were unable to eliminate all of the high frequency signals doing this and there remains a prominent high frequency tone in the output audio.

Another limitation is the camera quality. Our current implementation has a few issues with the displayed video. The ones that are the most obvious are the splitting at the beginning of the incoming video frame, the duplication of the incoming image, and the rows of entirely black pixels. These issues negatively impact user experience and would need to be addressed before moving on to our desired extensions.

One of the biggest limitations for our project was the fact that we often needed two FPGAs to test our implementation. Because our project is heavily reliant on ethernet to transmit data between FPGAs, it was impractical to write a simulation over the duration of our project. This resulted in us spending many hours together in the lab, since we only had two FPGAs. If we were to undertake the project again, we would likely ask for an additional loaner FPGA to allow for more asynchronous development. It was often difficult to schedule times when both of us were free, resulting in multiple trips to trade off FPGAs.



Video and Audio Effects - Peyton

As stretch goals for our project, we proposed optional video and audio effects. The goal was to include at least one effect for video or audio, but we were able to implement three visual effects and one audio effect. If there were more onboard BRAM, we would have integrated more effects.

During a video call, a user can use four switches to stack effects onto their transmitted video. We opted for switches so as to allow the user to mix and match effects, yielding more combinations and a more diverse chatting experience.

The four available effects are a Santa hat which is overlaid onto a user's video in a fixed position, a black and white video effect, invertible video colors, and a random noise effect for audio transmission.

The Santa hat is produced by drawing a 32x32 picture blob from a COE file on the screen. To improve the quality of experience, we built-in very simple background removing logic. The hat image is a contour of mostly red and white pixels, but the COE file we generated makes it appear as a square, with the hat positioned against a black backdrop. In the event that a hat pixel is supposed to be drawn, but is all black (12 bits of 0), we instead draw the video pixel for that point.

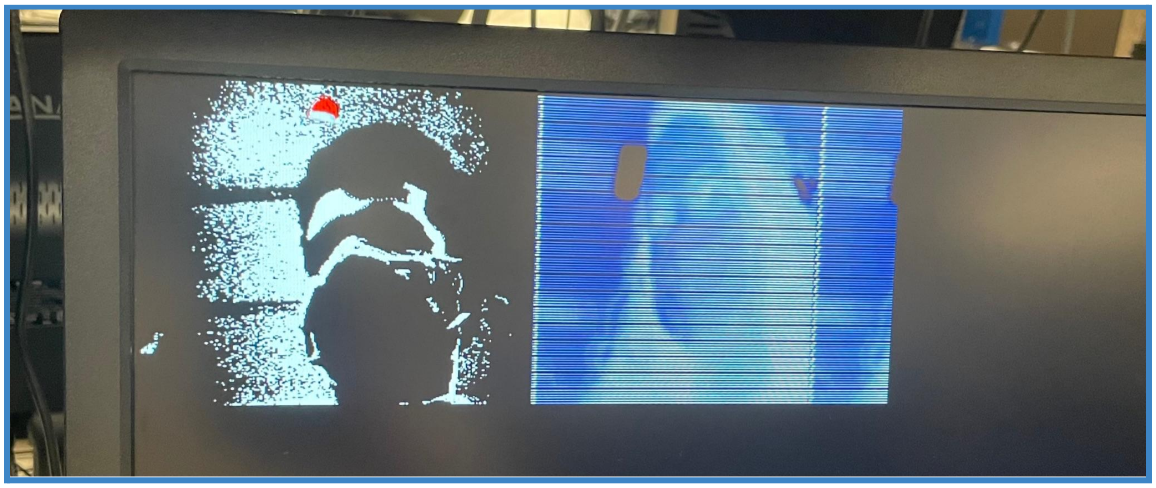
The black and white effect sums red, green, and blue pixel data, checks if the sum is greater than or equal to 24, and uses this boolean value as black or white. This is a simple, yet efficient bitwise algorithm which performs a pseudo average of color data. The maximum number of colors for a 4 bit value is 16 (the maximum sum for three pixels is 48), therefore performing the comparison yields a 1 bit value which can be replicated to display a 12 bit pixel.

The inverted colors effect simply bitwise negates pixel data. It should be noted, however, that the inverted colors can be combined with the black and white / hat effect. This was accomplished by multiplexing the effect toggle switches with each possible case in the drawing logic.

The random noise effect adds audio data from the receiver's microphone to the received audio transmission before filtering. This can sometimes help with quantization, but also produces an interesting tone.

To display the various effects on the receiving end, we opted to include a small effects header within the ethernet payload. The effects header is a 4 bit value which uses one-hot encoding to indicate the effects the sender is applying. Using this header, we then reconstruct the effects on the receiving end. This means that the unaltered camera data is still sent via ethernet.

We opted for this client-side effects approach as it allowed for increased modularity. The camera controller packages all video data while the audio controller packages audio data. This abstraction required the fewest number of modifications to the baseline video chatting platform.



Major Challenges - Jade

One challenge that we encountered throughout this project was the impracticality of using simulation to test the functionality of many of our modules. In fact, the only module that we were able to use simulation to test was the display fsm. Since most of our modules required inputs from external components (mic, cam) we would have had to produce a large amount of data to mimic the real time gathering of data from these sources. This wasn't as much of a hindrance to simulation though as the output of the module was either too large (ethernet) or difficult to translate into bitwise interpretation (mic, cam). Not being able to test our major modules in simulation meant that we lost a lot of time generating bitstreams for every little thing that we wanted to test. At the worst, these bitstreams were taking between 45 minutes and an hour to generate.

The time lost to having to generate bitstreams for every potential fix ended up being a major hindrance. When testing the camera module, we would comment out the audio module to decrease synth and implementation times, yet the bitstream generating still took between 10 and 20 minutes, so we were really only able to test one to three iterations in an hour and small mistakes like improperly setting up a BRAM or forgetting to change one variable wasted a significant amount of time.

Due to the large amount of time we had to dedicate to generating bitstreams along with the difficulty we had with video replay, we were concerned that we would not complete our MVP. The second major challenge that we had was replaying audio. In our initial proposal, we anticipated needing a frame buffer on both the sending and receiving FPGAs (ie. two frame buffers on each FPGA), but due to concerns about available memory, we started out attempting to directly stream video with a 25 MHz XVGA clock rate. Initially, we packed just under a line of pixel information into each payload. To try and fix what we believed were synchronization errors, we eventually added payload buffers that scaled from a depth of two to two hundred forty. During these iterations, we were never able to see more than one line of camera data at a time.

The next place we looked for errors was in the drawing logic. To mitigate potential issues with the payload being slightly less than a line, we increase the payload size to 320 pixels worth of bits. This change did not immediately produce a better result, but we kept it to simplify the printing logic. Finally, we decided to print on a 65mhz clock rather than a 25 mhz clock. With this change we were able to get enough of the video packet to print in order to have an acceptable looking video.

Future Extensions - Jade

We want to expand upon the capabilities we already have in the future by going to wireless sending of camera and audio packets. In order to allow for larger distances between physical FPGAs, we would use WiFi to send rather than ethernet.

A second extension is the addition of multiple users. Currently, due to the ethernet cable, we are limited to sending only between a single pair of FPGAs. If we implement WiFi, we also want to allow the user to call more than one other person. Along with being able to connect with more than one person, a desired functionality would be to initiate a call with multiple users at once, so that a single call may have 2+ people in it at a time. Another extension we would like to implement is motion tracking with the video overlays. With this enabled, visual image overlays like the santa hat would move as the user moves to ensure proper placement.

A final extension is the development of simulation software for ethernet transmission. The biggest challenge of our project was the fact that we had to test our system in hardware. The delay we experienced while waiting to generate bitstreams slowed the project's progress significantly. To build a simulation, we could use a packet capturing software such as Wireshark to create sample data. The sample data could then be fed into a testbench, where we could monitor and verify the output. This would be an easy way to catch bugs and sanity check our system design.

Summary – Peyton

FPGA-Time is a hardware-based communication system which enables two users to chat through a video call. Users are able to communicate across ethernet by simply connecting their FPGAs together and “dialing” each other, similar to Apple's FaceTime. It supports visual and audio effects in real-time with high throughput and low latency, delivering a smooth chatting experience.

Code

GitHub Repo: https://github.mit.edu/pshields/6.111-Final-Project
HTML : In team project code folder.