Live-Feed Video Transmitted over the Network

Michael Lee, Kenny Martinez, Carlos D. Nunez, James Phan, Patricio Tapia
Columbia University, New York, NY 10027
Department of Electrical Engineering & Computer Engineering
{ml4719, km3734, cdn2128, bnp2113, pet2119} @columbia.edu

Abstract – **This project is an implementation of a live stream between one camera and one viewer. This is accomplished by having two DE1 FPGAs communicate with each other over a local network. The "transmitting" board displays data from an OV7670 CMOS camera to a connected VGA monitor and sends a collected frame buffer over the network as a series of UDP packets. On the other end of the network, a "receiving" board takes and organizes this data, eventually displaying this data itself on an additional VGA monitor.**

I. Project Overview

Facilitated by the internet, millions today around the globe enjoy live streamed content produced from all other parts of the world. Despite its popularity and extensive use, the process of converting a camera feed into a comprehensible video on the consumer-side is complex and necessarily complex as higher resolution standards, such as 4K, gain traction. As higher resolution standards are called to encode streamable and real-time content, engineers will be pressed to find ways to maintain and increase data rates between devices to maintain acceptable refresh rates. In an effort to more deeply understand this problem and appreciate its challenges, our group was motivated to create a system for sharing real-time video streams on a local network. In doing so, we've found that this application is so commonplace in modern everyday life.

The goal of this project is to use two DE1-SoC FPGAs and communicate the live-feed of an external camera from one to the other. A successful implementation of this idea would result in the data from the camera being displayed to two external VGA monitors - one at each of the two transmitting and receiver boards.

On the transmitter side, data is saved to an instance of Block Random Access Memory (BRAM) which acts as a "frame buffer" which stores all the pixel values from one frame of video. Simultaneously, each byte of camera data is sent to a VGA module to display this information in real time. To simplify the design, our team made the decision early on to collect and display only in black and white by sampling the YGB channels of the camera. The persistent copy of data inside the frame buffer is sent in packets of 508 bytes per UDP packet (to minimize packet loss) to the receiving board. This is done through a userspace program which reads the data stored in BRAM through a driver that our team has written.

The transmitter has a complementary userspace program that receives the UDP packets containing parts of the frame buffer and stores that in memory. When it collects a full frame, the program writes into an instance of BRAM in the receiver board. After this has been completed successfully, the hardware of the receiver board accesses this data and displays it to the receiver's VGA display.

II. Milestones

As part of creating our design, milestones were set by our team as part of the project proposal. The milestones are presented below for reference.

Milestone 1 (25%): Interface Camera with the FPGA using I2C protocol. The goal for this milestone is get the FPGA communicating with the camera by following the I2C protocol so we can begin processing the raw data coming from the camera.

Milestone 2 (50%): Networking. For this milestone the main goal is to sort out the networking portion of the project. At this point the fpga should be able to communicate with a computer in the network and be able to start to send the pixels that have been processed.

Milestone 3 (75%): Achieve raw live-feed video on VGA display. For this milestone, our project should be capable of displaying a very raw version of the live-feed. The display should be recognizable but will probably contain glitches and issues that make it behave weirdly.

III. System Block Diagram

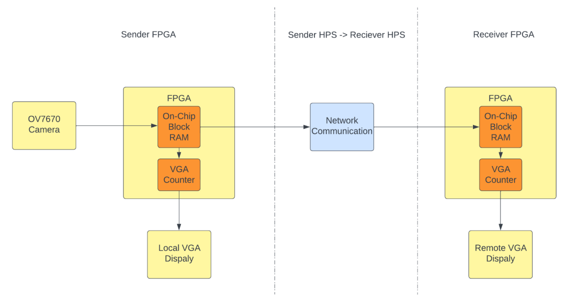


Fig. 1: System Block Diagram

IV. Resource Budget

One of the largest limitations present throughout the design was memory usage. While the DE1-SoC FPGA does have the ability to access large reserves of memory stored off-chip, this option presents additional challenges. One of the notable challenges is that in order to access the FPGAs SDRAM it requires one to deal with different clock domains. This challenge of crossing clock domains was troublesome enough in the transfer of data from the FPGA to HPS that it was chosen to instead exclusively work off of the FPGA on-chip memory.

The decision to exclusively utilize on-chip memory saves our design from further having to deal with separate clock domains but comes at the cost of having to operate the design using limited memory resources. The DE1=SoC has a total of 4,450 Kbits of embedded memory available for use in our hardware designs. The design heavily relies on using a frame buffer to send over data either to the network or to the VGA display. The next question presented by this limitation was regarding how large to make the frame buffer. This required some calculations that considered the resolution of VGA, the pixel depth, and bandwidth restrictions which are depicted in table 1.

Table 1. Memory and Bandwith Usage

Option	Frame Size	Frame Rate (fps)	B&W Bit Depth	Frame size (Mbit)	Bandwidth (Mbit)
1	640 x 480	30	8	2.4576	73.73
2	640 x 480	30	4	1.2288	36.86
3	640 x 480	15	4	1.2288	18.43
4	320 x 240	30	4	0.3072	9.22
5	320 x 240	15	4	0.3072	4.61
6	320 x 240	7.5	4	0.3072	2.3
7	320 x 240	5	4	0.3072	1.54

Note: Calculation of memory and bandwitch usages using differnt pixel depths and frame rates.

As a result of these calculations it was determined that option 1 was ideally the image quality that was desired for the display output from the VGA display. As can be shown in the table above, this option is one of the more memory costly options but our group felt that an 8 bit pixel depth provides a nice image whereas smaller pixel depths resulted in poor quality images which can be seen in the figure below.

This option was also chosen due to the fact that the resulting bandwidth was acceptable due the FPGAs 1 gigabit ethernet port. It is worth noting however that due to the frame's resulting size it was only possible to store a single frame in memory.

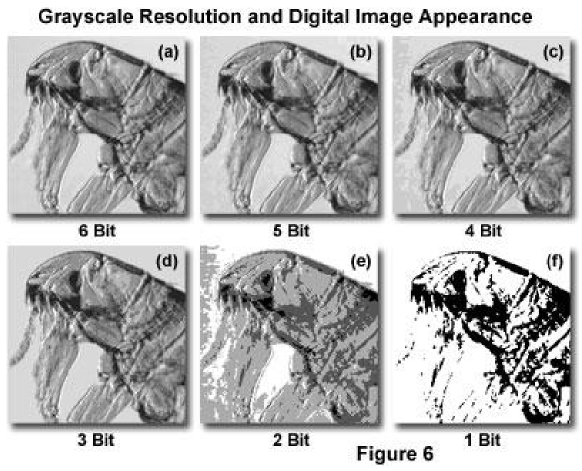
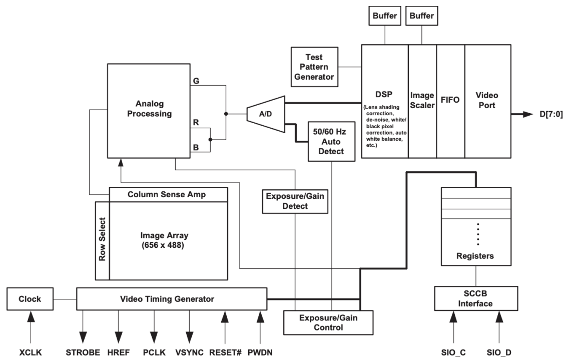


Fig 2: Image shows the effect of pixel depth on the image quality in black and white.[1]

V. Design: Hardware

OV7670 Camera:

We use an OV7670 VGA CMOS Camera Module as our input image. Luckily, the camera module comes with image processing so that the data we receive from the device in hardware is already conditioned. We can additionally specify certain channels and parameters. We've found that if we want to send a grayscale image, it is sufficient to use the yellow channel using a YCBCR standard which gives us the luminance of the image. To do so, we sample every other byte of camera output. It's to be noted that because of the image processing module on board, every pixel we receive from the camera comes in two bytes.



[1] Grayscale Resolution https://hamamatsu.magnet.fsu.edu/articles/digitalimagebasics.html

Fig. 3: OV7670 Block Diagram[2]

Unfortunately, using this camera module is not as simple as "plug-and-play". In order to receive any sort of output, we must first give the camera module an initialization sequence. To further complicate this task, the module does not speak a widespread standard, instead using a protocol called SCCB which differs from I2C in a few important respects such that communication is not as simple as using an I2C peripheral. We've found and verified an initialization module for the OV7670 camera which functions essentially as a state machine, configuring each of the camera's 201 configuration registers.

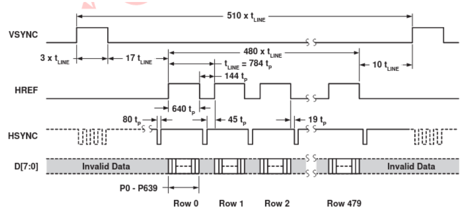


Fig 4. Timing diagram of the OV7670 camera

After configuring the module, we receive a few signals, not least of which are the VSYNC and PCLK outputs which pull high at the beginning and end of camera data and denote the timing of incoming camera data, respectively.

Although the pixel clock of the camera is 25Mhz, because we sample every other pixel, our effective rate is 12.5Mhz.

Transmitter FPGA Main Logic:

The transmitter FPGA main responsibility is taking in data from the camera and storing it into BRAM and then simultaneously displaying that data in BRAM to a VGA display and sending the frame to HPS via the Avalon bus. We utilize the VGA counter module in lab3 and modify it to output the endOfField signal as well. All in all, besides the camera initialization module, the top module relies on three finite state machines to function.

The sequence of events proceeds as follows. On start-up, we initialize the OV7670 camera as described above. The first state machine keeps track of the camera pixel count. Based on the HS, VS, and PIXEL clock from the camera, we increment the pixel count every time we get a valid byte. This pixel count is then reset to zero when we go from VS low to high. This pixel is needed later to decide when to write to BRAM again after reading from it.

The second state machine, shall to be referred to as the write FSM from now on, controls the writing operation to BRAM. It almost mirrors the first state machine. It differs in one additional state where it waits for read FSM to be done reading from BRAM to the VGA display. When we are done reading from BRAM, we need a way to know where in the frame the camera is outputting, which is why we keep track of the pixel count in the first FSM. In this write FSM, after waiting for the read FSM, we have to wait again for pixel count to reset and now we're in sync with the first FSM and we can begin to write a complete new frame to BRAM.

In the last FSM, we wait for the write FSM to finish writing. Then most likely the VGA counter is somewhere in the frame and not necessarily in the beginning of the frame. The read address is determined by the hcount and vcount calculated by the VGA counter. When it's done reading this 'partial' frame, we will reset the read address and increment the read address as the pixel clock is going as long as the VGA counter is in the active pixel area. Doing this will result in a less jittery video feed.

State Machine - Transmitter:

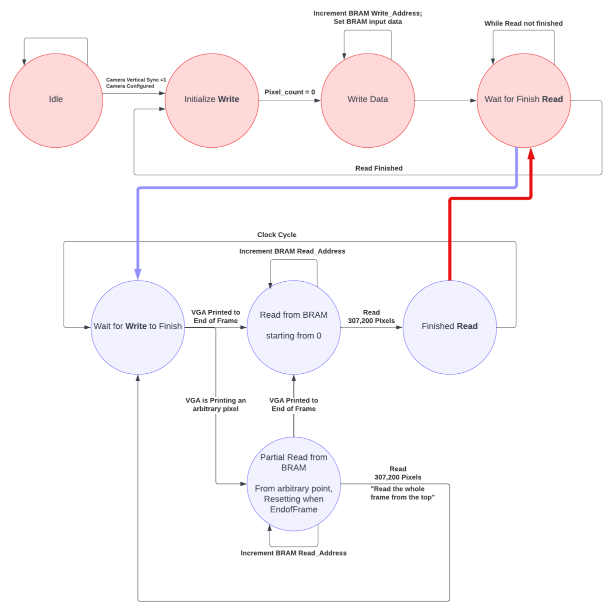

Idle
Camera Vertical Sync =1
Camera Configured
Initialize Write
Pixel_count = 0
Increment BRAM Write_Address;
Set BRAM input data
Write Data
While Read not finished
Wait for Finish Read
Read Finished
Clock Cycle
Increment BRAM Read_Address
Wait for Write to Finish
VGA Printed to End of Frame
Read from BRAM starting from 0
Read 307,200 Pixels
Finished Read
VGA is Printing an arbitrary pixel
VGA Printed to End of Frame
Partial Read from BRAM From arbitrary point, Resetting when EndofFrame
Read 307,200 Pixels "Read the whole frame from the top"
Increment BRAM Read_Address

Fig 5: Read-Write State Machines - Transmitter FPGA

BRAM:

In an effort to keep things simple, we use an inferred BRAM implementation introduced during lecture. We have an address for every pixel in a frame (640 x 480 = 307,200) where each pixel is represented by 8 bits. It is convenient to have the system set-up this way since it allows us to avoid using SDRAM and its associated challenges and is made possible thanks to the bit sizing

[2] OV7670_2006, http://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf

chosen allowing us to fill one complete frame. Furthermore, the address indexing into memory also serves as the pixel count.

VI. Hardware / Software Interface

The interfacing of the design's hardware and software components was done by utilizing the skeleton code provided for lab 3. Not only did it provide a base code to build upon but it also had an existing hardware module for interacting with the VGA display. The interfacing was done by editing the lab 3 driver to be able to receive / transmit data necessary to send over our frame.

On the transmitter FPGA, the HPS will just read from the FPGA, 32 bits at a time. We embedded the pixel count (19 bits) and the data (8 bits) so that we could build a frame buffer in software properly. For whatever reason the pixels skip when we read, we could still retain the spatial integrity of the image this way.

As for the receiver FPGA, the lab 3 code was modified to be able to write a 32 bit piece of data over the avalon bus as well. Contained within the 32 bits were 8 bits containing the color of the pixel and 19 bits containing the write address which were both fed from software. The form of this data which can be viewed in the figure below.

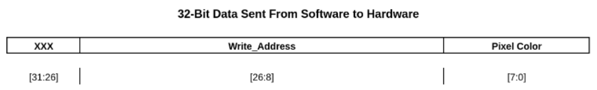


Fig 6. Bit ordering of data sent to receiver hardware from software

VII. Software

Transmitter FPGA:

We utilize some boilerplate UDP socket programming. We determine that sending a packet of size 508 bytes will result in the least amount of packet loss so that's how big our packets are. After getting a whole frame from hardware and storing it in a frame buffer, we send it over the socket, 508 bytes at a time until the last one, which is 368 bytes.

Receiver FPGA:

Similar to the transmitter FPGA, UDP socket programming was used. The receiver FPGA waits for a connection from the transmitter and once it receives a connection the receiver writes the packet's data into a buffer that is of size of a frame which is 640x480 bytes. The receiver continuously writes to the buffer until it is full. Once the buffer is full the buffer is then saved into the BRAM which then gets sent over to the VGA display. This process repeats until either the transmitter or receiver stops running.

VIII. Results

Our submitted design accomplishes many of the milestones we set out to complete with some complications present. Our design at the time of presentation successfully managed to set-up the camera for black and white video capture. The sender FPGA was able to process and display the image using a VGA module repurposed from lab 3 to be able to achieve live-video feed. Our design was also capable of communicating over the network to a receiver FPGA utilizing a UDP network protocol written into the FPGA HPS system. Software-Hardware interfaces were also successfully implemented utilizing drivers that were set-up to transmit the pixel color data. The receiver FPGA essentially utilized the same hardware resources of the sender with some modifications. These modifications accounted for the fact that data was instead received and stored to on-chip memory and then displayed onto a remote VGA display. Unfortunately our design isn't without complications that our team was unable to resolve and a later section is entirely dedicated to discussing these.

IX. Team Member Contribution

Michael Lee:
- Receiver Hardware
- Receiver Hardware/Software Interface
- Receiver Software
- Report

Kenny Martinez:
- Networking
- Receiver Hardware/Software Interface
- Receiver Hardware
- Receiver Software

Carlos D. Nunez
- Receiver Hardware
- Receiver Hardware/Software Interface
- Receiver Software
- Powerpoint/Report

James Phan:
- Interfacing Camera
- Sender Hardware
- Sender Hardware/Software Interface
- Sender Software
- Report

Patricio Tapia:
- Interfacing Camera
- Networking

- Sender Hardware
- Sender Software

X. Complication / Lessons Learned

During the creation of our design project we encountered a few complications along the way. One of the most notable issues present in our design is the flickering that is present in our local live-feed. The exact cause of this still remains unknown but a few possible solutions were brainstormed such as possibly reducing the memory used per frame in order to store multiple frames and hopefully resolve the flickering. The second notable issue was that in the process of sending data to HPS from the sender FPGA, the frame was not being correctly transmitted resulting in pixels being lost . This resulted in a poor quality image that can just barely be recognizable from the original image. The last known issue that negatively impacted the design was the networking. When sending a frame over the network to the receiver FPGA, the frame would arrive distorted. A proposed solution for this was that instead of just sending the pixel color data, the frame should have also sent information regarding its address. This would have made it easier to build the frame on the Receiver even if information arrived out of order since the address could be referenced.

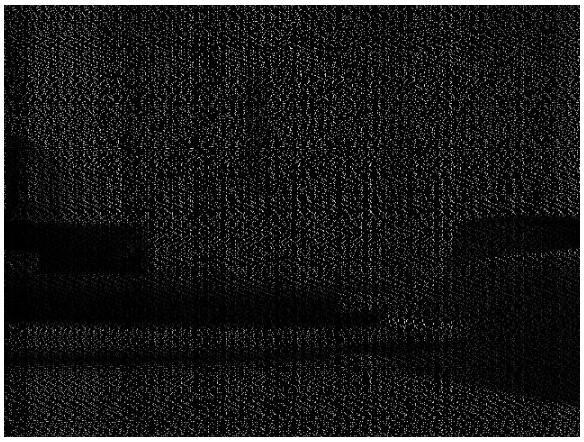


Fig 8. Depict issues present when transferring over pixel data. As a result the original image is barely recognizable.

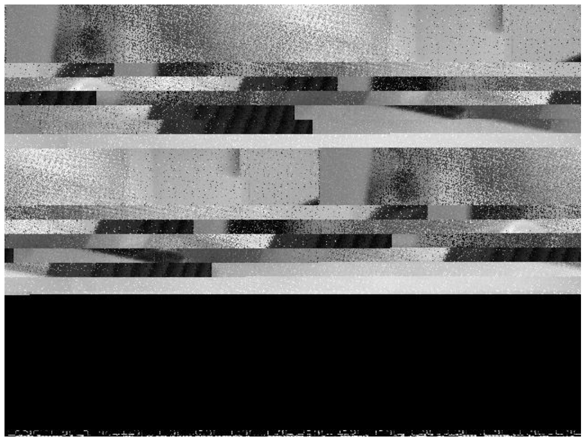


Fig 9. Distorted frame received over the network on receiver fpga.

Despite these setbacks our team was grateful for the opportunity to learn more about such a readily available feature as transmitting live-video feed. This project left all of us with a great appreciation for this common feature due to the difficulty and struggles we faced in the many parts of the project from interfacing the camera to sending even just sending one frame over the network. The biggest advice our team can give to teams in the future is to plan well in advance. Careful planning can make a major difference and avoid the pitfall of trying to create a design that is physically unrealizable. In our case some limitations didn't become apparent until late into the project. Lastly, we cannot stress the importance of the fact that there are many resources available, our project was by no means revolutionary and parts of it have been done and handled in much more efficient ways. Teams should research as much as they can and become aware of all the options available to them as well as not be afraid to seek help from the teaching assistants or Professor Edwards.

XI. Conclusion

In conclusion, our design was able to set-up the main components including the camera interface, frame memory storage, vga output and networking components. Our design was capable of displaying a live-video feed as well as capable of communicating over a UDP network protocol. Unfortunately as was discussed in our report, our design is not free of bugs that cause issues from flickering video to distorted video transfer. Future work should be focused on redesigning the memory storage and embedding address with the pixel color information to prevent these complications.

XII. Source Code List

Featured below is a list of the files that were created/modified for use in our project. Files that were created by qsys or quartus were not included in the list. Some may be found in our tar file that was uploaded such as the top module just for reference.

Sender FPGA Hardware:

- vga_ball.sv
- cam_vga.sv
- clk_div.sv
- OV7670_config.sv
- SCCB_interface.v
- vga_counter.sv

Sender FPGA Software:

- hello.c
- sender.c
- cam_vga.c
- cam_vga.h
- vga_ball.c
- vga_vall.h

Receiver FPGA Hardware:

- vga_ball.sv

Receiver FPGA Software:

- hello.c
- server_socket.c
- vga_ball.c
- vga_vall.h