ECE532 - Final Group Report

VidAccel: FPGA Accelerated Video Streaming

Group: 7

Students: Tianyi Zhang
Kanver Bhandal
Bruno Almeida

Submission Date: April 14, 2021

1.0 Overview

Our project is to build a video streaming platform, essentially something similar to Netflix but much smaller in scope. For our project, we used FPGAs to accelerate the encoding and decoding part of video streaming since this takes a significant portion of time on a CPU.

Our team is interested in video streaming since it is a clear application where FPGAs are used in the real world and have advantages over traditional computing devices such as CPUs. In principle, given more time than this course allows, this approach to video streaming could scale to many more devices and have clear benefits. We have also been researching encoding and decoding algorithms as they can compress a large amount of information very efficiently, so we wanted to learn more about how they work. Additionally, we all watch videos online but sometimes buffering or a drop in quality occurs. It would be interesting to try to make our own streaming platform where we minimize the occurrence of buffering or quality drops by having a lightweight network and frame encoding/decoding.

Similar work to our project does exist. Actually, part of our project has already been implemented by using FPGAs to perform video encoding and decoding. Intel provides an HEVC encoder block for their FPGAs [1]. Their example use case for the block is for video streaming from a server. As the latest video encoders are quite complex, such as H.265, it would take a large amount of time on a traditional CPU to encode, but an FPGA can be used to accelerate the encoding. The advantage of their solution is that the server can be much more responsive as the CPU does not get delayed by the encoding. However, the server would still have to go through the traditional OSI model to deliver the data to the user which could potentially be a bottleneck on performance as a custom network connection to the client could be much faster.

Our project's goal is to provide the client with the frames of their video and display those frames as a live video or stream. Our network will use colour cell compression to send the frames to the client. By encoding the RGB frames and using a lightweight network stack, we plan to reduce the bandwidth requirements and provide the user with a faster network connection to stream the video. To start, the video will be 160x120 pixels to ensure the video transfer speed, and there will only be one PC client. As video encoding and decoding with FPGAs already exists, the only advantage of our approach is that we can build a lightweight network to send the video frames to the client. This allows our approach to provide the client with a faster network connection compared to existing approaches as less network processing is required.

Figure 1 is the block diagram of our final design. To stream a video, a user can provide the video they would like to stream to the PC server, which would then be converted into RGB frames and sent to a FPGA server via FPGANet. The video frames would be encoded and stored on a SD card for long term storage. When the video is ready to be streamed, the encoded video would be sent to our FPGA client and decoded back into RGB frames. These frames would be sent to the PC client which allows the end user to watch the stream. The FPGA client can also receive commands from the user to start or stop the stream.

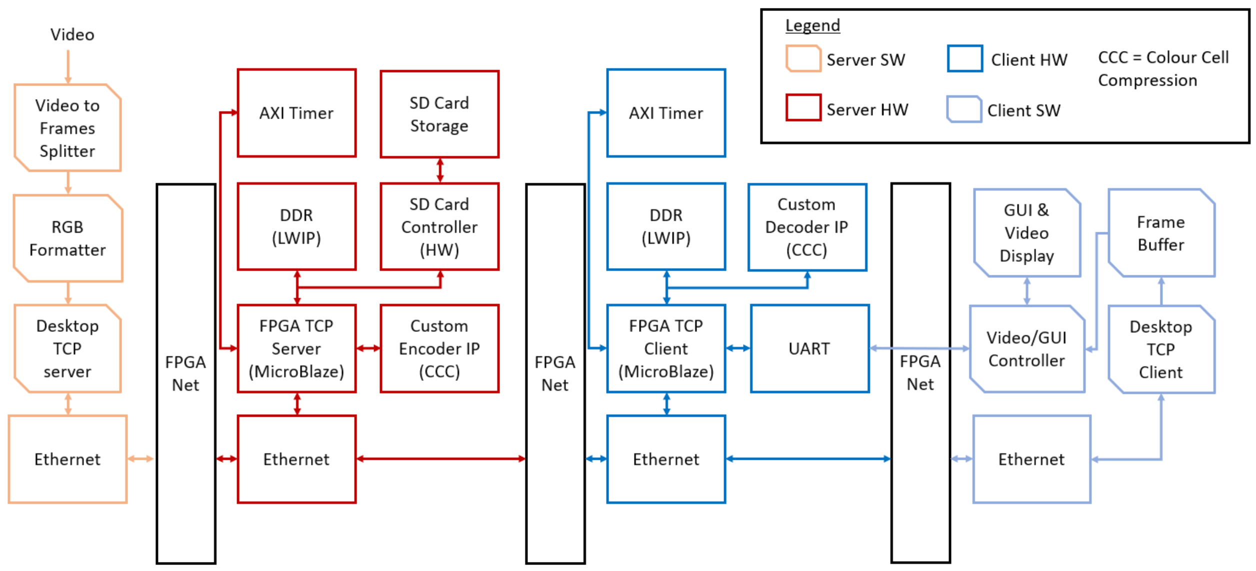

Figure 1: Block Diagram

Tables 1 and 2 list the IP blocks used, modified, and created in our design. More detailed descriptions are included in Section 4.0.

Table 1: List of Hardware IP Blocks

IP	Source	Description
AXI 1G/2.5G Ethernet Subsystem	Used - Xilinx	Default Xilinx block. Used to send and receive video frames over FPGANet.
AXI Direct Memory Access	Used - Xilinx	Default Xilinx block. Used to move data from/to memory.
AXI SmartConnect	Used - Xilinx	Default Xilinx block. Used to access the Memory Interface Generator.
AXI Timer	Used - Xilinx	Default Xilinx block. Used for performance measurements.
AXI Uartlite	Used - Xilinx	Default Xilinx block. Used for logging and debugging.
ccc_codec_ip_v1.0	Created	Combined encoding and decoding algorithms.
Clocking Wizard	Used - Xilinx	Default Xilinx block. Used to generate clock signals.
MicroBlaze Debug Module	Used - Xilinx	Default Xilinx block. Used for JTAG-based debugging of MicroBlaze processors.
MicroBlaze	Used - Xilinx	Default Xilinx block. Soft microprocessor core used to execute programs and interact with

		hardware.
AXI Interrupt Controller	Used - Xilinx	Default Xilinx block. Used for checking, enabling, and acknowledging interrupts.
AXI Interconnect	Used - Xilinx	Default Xilinx block. Used to connect one or more AXI memory mapped devices.
Concat	Used - Xilinx	Default Xilinx block. Used for concatenating interrupt signals.
Memory Interface Generator	Used - Xilinx	Default Xilinx block. Used to access DDR3 memory.
Processor System Reset	Used - Xilinx	Default Xilinx block. Used to reset the system.
sd_controller_0	Modified - MIT	Obtained from an MIT course website. Used for SD card initialization, read, and write.
SD_MYCODE_0	Created	Custom AXI Lite block. Used as a communication interface between the sd_controller and MicroBlaze.
Constant	Used - Xilinx	Default Xilinx block. Used to provide constant signals to the SD Card.

Table 2: List of Software IP Blocks

IP	Source	Description
Video to Frames Splitter	Created	Used to split a video into RGB frames.
RGB Formatter	Created	Used to resize video frames and convert them into RGB 888 format.
Desktop TCP Server	Created	Used to send PPM files to the FPGA TCP Server.
FPGA TCP Server	Created	Used to receive un-encoded video frames, encode them, store the encoded frames, and send them to a FPGA TCP Client.
FPGA TCP Client	Created	Used to receive encoded video frames, decode them, and send the decoded frames to the Desktop TCP Client.
Desktop TCP Client	Created	Used to receive decoded video frames from the FPGA TCP Client, and write the frames into the

		Frame Buffer.
Frame Buffer	Created	A collection of video frames. Used to ensure smooth playing of videos.
Video/GUI Controller	Created	Used to display frames on the GUI, and receive logging and debugging messages from UART.
GUI/Video Display	Created	Created using OpenCV and displays the video. Allows users to pause/start the video.
OpenCV	Used - Intel	Image processing library.
Socket	Used - Python	Networking library.
PySerial	Used - Python	A library for communicating to serial devices.
_thread	Used - Python	Low level Library for generating thread.
LWIP	Used - Xilinx	Lightweight networking library.
UARTLite Driver	Used - Xilinx	Xilinx provided driver to support UART.
AXI Timer Driver	Used - Xilinx	Xilinx provided driver to support the AXI Timer block.

2.0 Outcome

Overall, we accomplished the large majority of our originally planned functionality from the proposal. The core functionality was accomplished, in which an RGB frame from the Desktop Server was sent to the FPGA Server, encoded (compressed) from RGB to CCC, sent to the FPGA Client, decoded (decompressed) from CCC to RGB, sent to the Desktop Client, and displayed on the PC screen GUI.

We originally planned to send 640x480 frames, hopefully at 24 FPS (frames per second), but during testing, we discovered the FPGANet network was far too slow to accomplish this. In the end, we sent 160x120 frames. Even then, it was too slow to display the frames live as they were being transferred, so we decided to transfer all the frames of the video at once (137 frames) and buffer them on the Desktop Client software, then display the frames of the video quickly to look like a smooth video. We added the Frame Buffer block to the system block diagram as part of the Desktop Client to save all frames before playing them back.

Originally, we planned to have the Desktop Client send UART commands to the FPGA Client to start and stop the video stream (i.e. the FPGA Client would pause requesting new frames from the FPGA Server) since it would be streaming and displaying frames live. Given our change to transferring and buffering the entire video before displaying it, it no longer made sense to use UART for the Desktop Client to tell the FPGA Client to pause the video stream. Instead, the Desktop Client software could just pause displaying saved frames from its Frame Buffer. Therefore, we instead decided to use UART to have the Desktop Client PC receive debugging log messages from the FPGA Client and display them on screen to observe what the client MicroBlaze processor was doing live.

We originally planned to use the Xilinx AXI Quad SPI block to interface the MicroBlaze with the SD card on the FPGA Server, but this IP core could not toggle the SPI chip select (CS) signal and the data on the same clock edge. Therefore, we decided to use an SD card SPI controller module obtained from an MIT course website [2] instead to interface with the SD card. We designed our own IP core to provide an interface of AXI-Lite registers and interface with the SD controller module, therefore allowing the MicroBlaze to interface with the SD card.

We originally hoped to use the SD Card on the FPGA Server to save already encoded CCC frames to make it faster to retrieve them and send them to the FPGA Client if requested again. We had SD card reads/writes almost always working, but sometimes a couple of bytes were wrong when reading/writing a block. We could not figure out why this happened before the demo, but we speculated it might have been due to different clock domains (AXI at 100 MHz, SPI at 25 MHz), or it could have been a signal integrity issue on the board between the FPGA and SD card. This meant we could not trust 100% of the data read back from the SD card as it would have produced erroneous results in the frames. In our demo, we first demonstrated the SD card reads and writes worked for a small chunk of data (except a couple of bytes). Then, we moved into the main part of our demo to transfer a video of frames without using the SD card, which involved writing all encoded video frames to the SD card but never reading them back.

We originally planned to convert our TCP servers on both MicroBlazes and both Desktops to use raw IP packets with our own transport layer protocol after getting our core functionality working using TCP, hopefully to optimize performance and provide a higher throughput connection, but this did not pan out for multiple reasons:

1. It was not clear whether the MicroBlaze LWIP library was designed to allow us to use IP directly with the necessary callbacks, or if the IP functions were only designed for internal use.
2. If we wanted to implement raw IP in hardware rather than on the MicroBlaze, we could have used the TEMAC (Tri-Mode Ethernet MAC) module, which accepts AXI-Stream input and sends out raw IP packets. However, we learned that another group had been trying to use the TEMAC for most of their project timeline, but did not have much success due to complexity and a lack of documentation and examples. Given we would only have 1-2 weeks to implement IP, this was completely unfeasible for us to attempt.
3. We learned from another team that the DESL Windows computers require administrator privileges to access raw IP sockets and send/receive raw IP packets, which we did not have permissions for.

We hoped to compare the performance of TCP vs. UDP for network communication if we had time at the end of the project, but we did not have enough time for this at the end due to project integration taking much longer than expected. However, we did manage to collect performance measurements for TCP, frame encoding, and SD card writing:

- Desktop Server to FPGA Server: 8 Mbps
- Encoding frames: 6 Mbps
- Writing frames to SD card: 2 Mbps
- FPGA Server to FPGA Client: 2 Mbps

So overall, we achieved the core functionality of our project by streaming a video from server to client with encoding (compression) and decoding (decompression) using hardware blocks on both ends. However, we did not achieve all of the additional features we hoped could improve the project even further, such as the SD card for caching encoded frames, IP instead of TCP for better performance, and displaying the frames of the video live as they were being streamed and encoded/decoded. Our timeline and week-by-week priorities were reasonable, as we expected having to cut some features and focus on the core functionality if time was limited at the end, which ended up being the case.

If someone were to take over our project and continue it, we would recommend debugging the SD card issues with a couple of bytes being wrong in reads/writes. After this issue is fixed, it could be integrated fully with the FPGA Server. The code on that MicroBlaze would need to be modified to check when the client requests a frame whether that frame is in the SD card or not, and decide whether to fetch it from the SD card or encode it from scratch.

Our system could be improved by implementing a more efficient lossy encoding algorithm with a greater compression ratio (compared to our CCC which is 6x compression) to reduce the amount of data that needs to be sent over the network from FPGA Server to FPGA Client. Perhaps this would allow for higher resolution video and/or being able to stream and display video frames in real time. Alternatively, a lossless encoding algorithm could be used to improve image quality compared to our lossy CCC algorithm.

3.0 Project Schedule

Table 3 describes the original plan for each milestone (from our project proposal) compared to what we actually accomplished for each milestone (from our six milestone progress reports). Most milestones were met, but the later milestone schedules tended to diverge more from what we originally planned.

In hindsight, we should have allocated more time for packaging our IP cores in Vivado using AXI-Lite interfaces (encoder/decoder and SD controller). This turned out to be much more difficult and time-consuming than expected, especially when learning about the quirks of the tools.

Table 3 - Milestones

Milestone #1 - Feb 2			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	Modifying the project from the warmup demo so that it supports the Nexys Video board (DESL-B).	Modified the project from the warmup demo to support the Nexys Video board (DESL-B). The server and the client can communicate with their respective server/client Python script individually. The connection between the server and the client FPGAs has been set up and fully working.	Finished as expected.
Kanver	Using a Python script to convert a video file into RGB frames that is 640x480 pixels, where each pixel is 24 bits. 24 frames will be generated per second, which is the same as the standard movie frame rate.	Used a Python script to convert a video file into RGB frames that are 640x480 pixels, where each pixel is 24 bits. This was done using OpenCV and it runs on the DESL machines. Additionally, I made some progress on Milestone 2 as I've written a Python script to display the RGB frames as a video at 24 FPS. Helped support Tianyi for the Ethernet PHY issue.	Finished as expected, plus some progress on the video GUI for Milestone 2.
Bruno	Researching the color cell compression algorithm and high-level steps for implementing it.	Researched the colour cell compression (CCC) algorithm and developed a high-level plan for how to implement it in RTL.	Finished as expected.
Milestone #2 - Feb 9			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	Looking into the driver setup for SD card on Xilinx board and useful libraries for	Looked into the driver setup for SD card on Xilinx board and useful libraries for implementation. Read the documentation for SPI interfaces with SD cards.	Finished as expected, plus planned work for Milestone 3 in setting up the block diagram.

	implementation.	Implemented the block design using the Quad SPI IP and made all the connections, including pin assignments on the board. Successfully generated bitstream for the block design.	
Kanver	Writing a Python script to display RGB frames as a video and creating a GUI for user control input.	Implemented a GUI when watching the video. Callbacks were set up to execute when a button is pushed. The video viewer was modified so that the original frame is shown and not scaled up (this reduces the blurriness of the video).	Finished as expected.
Bruno	Implementing Verilog modules to encode RGB frames using the color cell compression algorithm and to decode them back to RGB frames.	Implemented Verilog modules for CCC encoding and decoding. Decided to use fixed-point math instead of floating-point math. Created the testbench module and got the behavioural simulation to run in Vivado with the encoder module instantiated (but no input data or debugging yet).	Finished as expected, plus some work on setting up Milestone 3 (testbench verification and debugging).
Milestone #3 - Feb 23			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	Setting up the library files and connections needed for the SD card interface.	Wrote a C program to initialize and test SPI in interrupt mode using read and write - fully functional. Researched into the communication protocols for the SD card using SPI. Implemented C programs for the communication, including initialization of the communication in SPI mode, transmitting data, and receiving responses. Debugging using SDK and ILA, as I'm not getting the correct responses from SD card yet.	Doing work originally planned for Milestones 4/5, working on C code and debugging.
Kanver	Implementing Python/C programs to send TCP packets between the various clients and servers. The three connections include: 1. Desktop server to FPGA server 2. FPGA server to FPGA client 3. FPGA client to desktop client	Desktop server and client works, can transfer data between them and display a video correctly. Desktop server to FPGA server works, can send an entire video to the FPGA Server and store it in DDR memory. FPGA server to FPGA client and FPGA client to Desktop client isn't implemented as time was spent optimizing the Desktop server to FPGA server connection. Also, on DESL some library packages were missing in OpenCV for Windows so I had to redo the GUI buttons.	Did not implement FPGA client as planned due to time spent optimizing the performance of the other network connections.
Bruno	Creating a testbench to simulate the encoding	Added the decoding module to the testbench. Set up the testbench to read	Debugged the modules as expected,

	and decoding between RGB frames and color cell compression. Verifying the results and debugging issues.	an input frame from a file, run the algorithms in the encoding and decoding modules, and save the output frame to a file. Debugged the testbench until it worked for a 4x4 pixel image.	but only working for a 4x4 image, not with large images as originally planned.
Milestone #4 - Mar 9			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	Implementing C programs to write data to the SD card and read it out.	Fixed the issue with MOSI using Quad SPI and was able to send the correct outputs to SD card, but couldn't get the correct response. Created and packaged a new IP for a custom SD card control module and created a Verilog module to communicate with it. Setup hardware (ILA) monitoring and started debugging the interface. Was able to get some responses from the MISO line.	Using the AXI Quad SPI block turned out to be a dead end, so pivoted to using the custom SD card controller module found online and implementing a custom module to communicate with the controller.
Kanver	Implementing software programs to send GUI commands to start and stop the video stream from the desktop to the FPGA client through UART.	Tested frame sending using PC Server -> FPGA Server -> PC Client, it works and the frame isn't corrupted. Was able to use Putty to communicate with FPGA using UART instead of the Xilinx SDK terminal. Also, was able to use UART in interrupt mode.	Did not get to implementing and testing the FPGA Client from Milestone 3 yet. UART mostly implemented, but not using Python yet.
Bruno	Integrating the encoder and decoder modules into the hardware system and verifying the results.	Debugged encoder/decoder testbench with 16x16, 80x60, and 160x120 frames until it worked. Created a new AXI-Lite slave peripheral IP. Modified the auto-generated code to support more registers and integrate our encoder and decoder modules. Ran Vivado synthesis, implementation, and bitstream generation.	Had to finish debugging of larger frames originally planned for Milestone 3. Packaging the IP core using an AXI-Lite interface with registers was more difficult than expected, so did not get to integrate the encoder/decoder with the larger system.
Milestone #5 - Mar 16			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	Verify functionality of C programs created in Milestone 4 by using them to read and write to the SD card and manually checking the output.	Finished implementation and verification of read/write on SD card using hardware. Modified the custom block for SD card interface to support newer versions of SD cards. Was able to receive expected responses from SD card. Used ILA to verify that SD card reads in what was written into it using custom Verilog IP	Finished what was originally planned for Milestone 5, but now with the alternate SD card approach.

		blocks.	
Kanver	Implementing the start and stop commands on MicroBlaze so that the FPGA server stops sending the video stream when the FPGA client sends a “stop” command.	Verified that we can send several frames from Desktop Server -> FPGA Server -> FPGA Client -> Desktop Client. The issue was incorrectly configuring how many packets we sent which was only noticed after a fair bit of debugging. Start and stop commands now work on the GUI and we can send start/stop commands to the FPGA Client over UART from Python.	Tested FPGA Client, originally planned for Milestone 3, and continued debugging remaining TCP problems. Start and stop commands implemented as expected.
Bruno	Modifying the TCP code for the MicroBlaze to support IP servers and clients using LWIP. Also modifying the Python code for the desktop to use IP instead of TCP.	Integrated our packaged IP core (AXI-Lite slave) into our project’s block diagram and generated a bitstream. Tested MicroBlaze access by writing 1 to the start register, then reading the done register until it became 1, which worked. Developed a new encoding/decoding testbench using the AXI VIP with the packaged IP core.	Decided not to pursue the original plan of converting TCP to IP due to time limitations in the project and technical limitations (discussed in Section 2.0). Had to continue debugging the codec IP core, both in AXI VIP simulation and on hardware using MicroBlaze.
Milestone #6 - Mar 23			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	Verifying the design by streaming a video file.	Added a custom AXI Lite interface to our Verilog for the SD Card so that MicroBlaze can write/read large files to the SD Card. Added handshake signals needed for the communication. Working on testing/simulating the interface.	Had to finish SD card IP packaging and integration.
Kanver	Verifying the GUI commands and the user control for the start and stop of video streams.	As the GUI commands now work, I moved onto improving our TCP performance and measuring it. For now TCP performance is good enough for use in a demo. Still working on implementing time measurements using AXI Timer blocks. I’ve been helping with the SD Card but that is taking a while due to dealing with SD Card and MicroBlaze/AXI-Lite interfaces.	GUI commands were completed. Decided to improve TCP performance and implement a new performance monitoring feature given the time available.
Bruno	Verifying the IP network of the design and debugging potential issues.	Re-tested the encoder/decoder IP core on hardware using the MicroBlaze. Only the first 4x4 block of pixels appears to be encoded and decoded correctly. Re-tested the core in simulation and discovered some registers in the encoder did not reset their values after each	Already decided not to use IP (from Milestone 5). Had to continue debugging codec IP core on hardware and in simulation due to remaining issues with

		frame, so I debugged it until it worked in simulation. Debugged the IP core in hardware by observing signals with ILAs while stepping through a MicroBlaze test program.	the core functionality.
Before Final Demo - Mar 30			
Member	**Original Plan**	**Actual Accomplishments**	**Discussion**
Tianyi	N/A	Tested the SD Card interface in simulation and verified the functionality of the design. Then helped Kanver integrate the SD card interface to work with his code on the MicroBlaze.	Was finally able to integrate with the main project.
Kanver	N/A	Added performance monitoring for some parts of our code using the AXI Timer blocks. Tested and integrated the system with help from Bruno and Tianyi. Verified that we can successfully send a video, encode it, decode it, and view it. Created the plan for what to demo.	Finished new performance monitoring. Was the primary integrator for the project.
Bruno	N/A	Recreated the codec IP core from scratch and regenerated the bitstream to fix Vivado issues and ensure it used the latest RTL files. Discovered timing failed and modified the RTL to achieve timing closure. Debugged the core with ILAs and a MicroBlaze test program. Assisted Kanver with integrating the codec IP core into the main block diagram and MicroBlaze code.	Had to deal with last-minute unexpected problems in the tools. Timing closure required a lot of time to debug and fix, then hardware testing went smoothly. Integration work with Kanver was mostly a matter of communication rather than technical issues with my block.

4.0 Description of Blocks

Below, in Figure 2, is our overall block diagram from Vivado that we used to generate our project. In this section, we will further discuss the hardware/software IP we used. Some of the IP blocks from Table 1 were combined into a more general section as they were auto generated when configuring certain blocks. For example, the AXI Interconnect block is described in Section 4.1.5 along with the MicroBlaze.

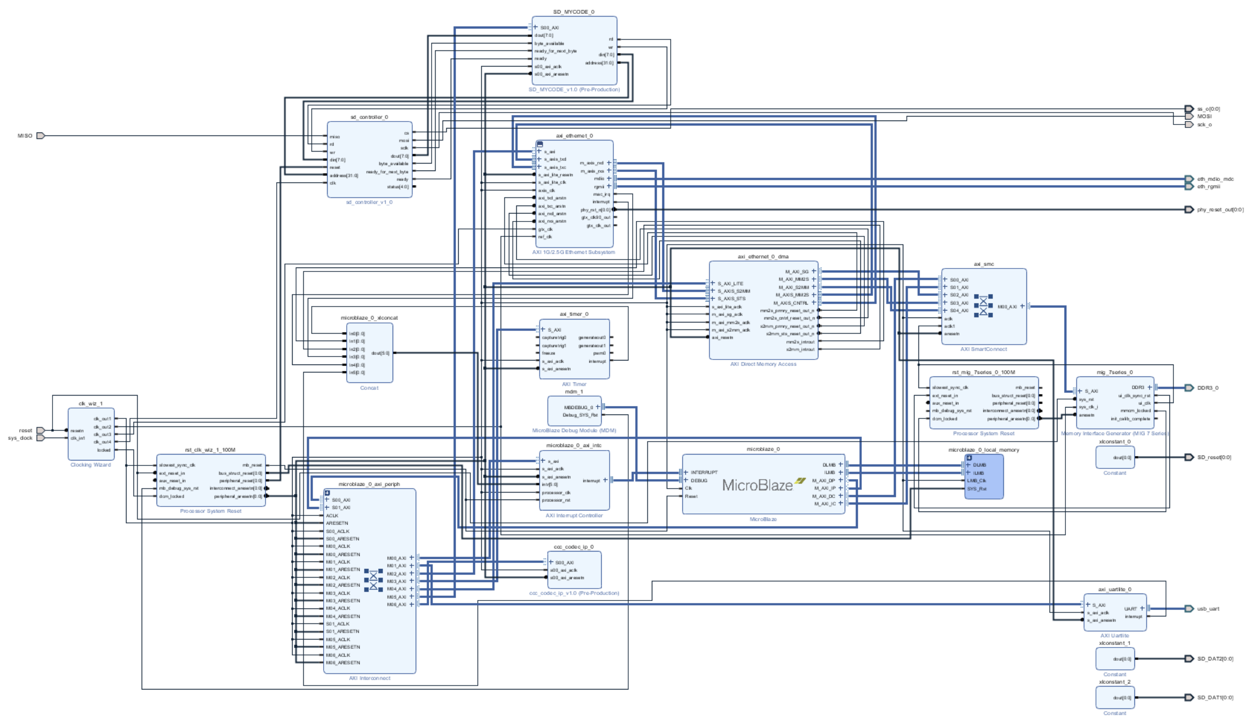

Figure 2 - Overall System Block Diagram from Vivado

4.1 Hardware IP

Below the hardware IP used in our project will be discussed.

4.1.1 CCC Codec

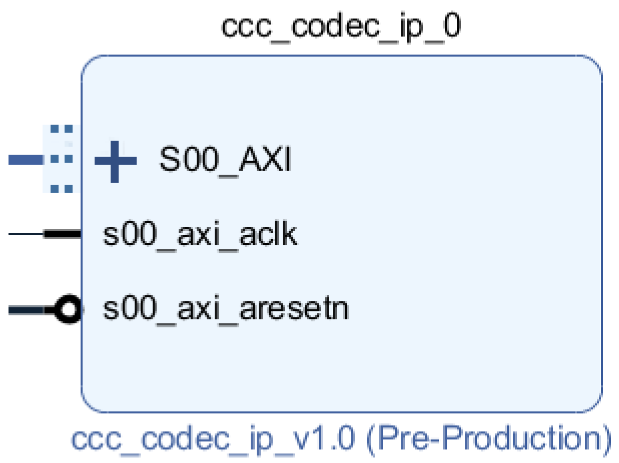

Figure 3 - CCC Codec Block

The encoder and decoder blocks are combined into a single IP core known as the CCC Codec (Figure 3). They were combined into a single core to make it easier for integration, so that we only needed one Vivado project, one block diagram, and one bitstream that works for both the FPGA Server and FPGA Client. The server's MicroBlaze software only uses the encoder part of the block, while the client's MicroBlaze software only uses the decoder part of the block. The IP core uses an AXI-Lite interface with a 32-bit data width (32-bit registers).

The encoder and decoder each operate on a single chunk of the frame, 4x4 pixels, at a time. The MicroBlaze writes a 1 to the start register to start the processing, then polls the done register until it is 1, which is set by the hardware block automatically when finished and reset by the hardware block automatically when the register is read by the MicroBlaze.

Each RGB pixel has a pixel depth (colour depth) of 24 bits, i.e. the R component is 8 bits, the G component is 8 bits, and the B component is 8 bits. Each CCC block (of 4x4 pixels) has 64 bits of data, i.e. 16 bits for the bitmap, 24 bits for the bucket 0 RGB mean, and 24 bits for the bucket 1 RGB mean.

This means our compression ratio is a factor of 6 as follows, using the number of bits for each 4x4 block of pixels:

compression ratio = (4x4 pixels x 24 bits/pixel) / (64 bits) = 384 bits / 64 bits = 6

4.1.1.1 Encoder

The encoder has 16 registers for RGB (input) data, 2 registers for CCC (output) data, a start register, and a done register.

The encoder performs the following algorithm:

1. Calculate the luminance of each pixel from the RGB input registers, which measures the light intensity, weighted differently for each of the red (R), green (G), and blue (B)

components. We implement this using fixed-point math because using the Xilinx floating-point core would have added much more complexity without much benefit.

luminance = ((77 * R) + (151 * G) + (28 * B)) / 256

2. Sort the 16 pixels in order from least to greatest luminance, keeping track of each pixel's coordinates in the original 4x4 block.
3. Group the 8 pixels of least luminance in bucket 0, and the 8 pixels of greatest luminance in bucket 1, to create the bitmap.
4. Calculate the mean of the RGB pixel values in bucket 0, and separately calculate the mean of the RGB pixel values in bucket 1.
5. Save the bitmap, bucket 0 RGB mean, and bucket 1 RGB mean in the CCC output registers.

4.1.1.2 Decoder

The decoder's operation is simpler than the encoder, and it does the reverse operation of the encoder to recover the RGB data from the CCC data.

The decoder has 2 registers for CCC (input) data, 16 registers for RGB (output) data, a start register, and a done register.

The decoder performs the following algorithm:

1. For each of the 4x4 pixels, check from the bitmap which bucket it is in.
    a. If bucket 0, assign the bucket 0 RGB mean to the pixel's RGB output register.
    b. If bucket 1, assign the bucket 1 RGB mean to the pixel's RGB output register.

4.1.1.3 Testbenches

First, we developed a testbench (ccc_codec_tb.sv) that instantiates both the encoder and decoder modules. The testbench reads in an input frame from a PPM (raw RGB image) file, populates the encoder input (RGB) registers with the frame data, starts the encoder, and waits for the encoder to be done. Next, it reads the encoder output (CCC) registers, transfers the data to the decoder input (CCC) registers, starts the decoder, and waits for the decoder to be done. Finally, it reads the decoder output (RGB) registers and saves the frame data to a new PPM file.

The designer compares the input image file and output image file to see if they match, although the output image file will have a reduced quality due to compression, as expected.

We used PPM files for the images because they simply specify raw 24-bit RGB values for each pixel and were relatively easy to read, decode, and write out, compared to more complex image compression file formats. Note that the DESL Windows machines did not have software to display PPM files, so we used a website [3] to convert them to PNG files for viewing on Windows.

This testbench tested just the core encoder and decoder logic RTL files, without any AXI-Lite wrapper/interfacing. It was run using Vivado behavioural simulation. Note this testbench assumes the input image is the same dimensions as the encoding/decoding hardware, so it only needs to run the encoder and decoder once each.

The second testbench (ccc_codec_axi_vip_tb.sv) is very similar to the first, with two key differences:

1. It tests the encoder and decoder logic with the AXI-Lite interface wrapper, i.e. the entire CCC Codec IP core. It uses a separate Vivado project with an AXI VIP (Verification IP) acting as the AXI-Lite master and our IP core acting as the AXI-Lite slave.
2. It can test input images that are larger than the dimensions of the encoding/decoding hardware, by running the encoder/decoder multiple times, once for each block of 4x4 pixels.

4.1.2 SD Card

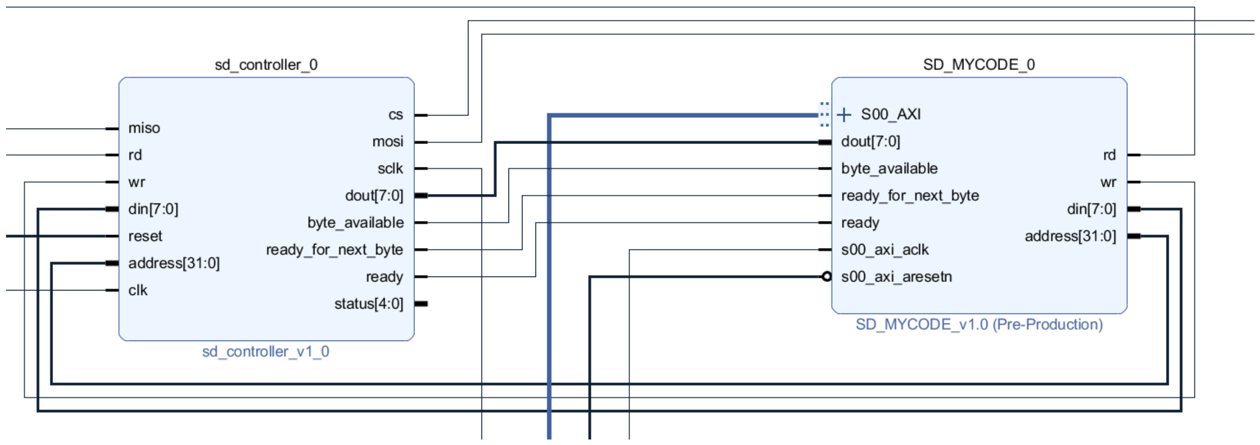

Figure 4 - The block diagram of the sd_controller and SD_MYCODE

In Figure 4, our SD Card blocks are shown. The sd_controller block was obtained from an MIT course website which includes the SPI mode initialization, read, and write commands required by SD card SPI protocol [2]. We modified this block so that it would work with newer versions of SD cards. Specifically, CMD8 was added after the correct response for CMD0 was received from the SD card in the initialization process. This block operates at a clock frequency of 25 MHz. The SD_MYCODE block is a custom AXI Lite block which serves as a communication interface between the sd_controller and MicroBlaze. The SD_MYCODE block was created from the Create and Package IP tool provided by Xilinx and modified to meet our interface requirements. It runs at the same clock speed as the rest of the AXI blocks in the system, 100 MHz.

The SD_MYCODE block includes 256 registers for data transmissions and handshake signals. How the block is used by the MicroBlaze is described next. Before reading or writing to the SD Card, register 250 must be read to check if it contains a 1. A 1 indicates the sd_controller is ready for a read/write operation. To write a block (512 bytes) to the SD Card, first the MicroBlaze writes the data block to registers 0 to 127 and provides a write address for the SD Card in register 251. Then it writes 1 to register 255 to start the write. The write is completed when register 254 is 1. Then the MicroBlaze must clear register 254 back to 0. For reading a similar process is followed, MicroBlaze provides an address to read in register 251 and initiates the read by writing 1 to register 253. Then it polls on register 252 until it reads a 1 which indicates the read is done. Then registers 0 to 127 contain the block of data from the SD Card. Also, register 252 must be cleared before another read can begin. Registers 128 - 249 are not used and are there for future expansion.

6 pins were mapped in the SDpin.xdc constraints file according to the board manual for the SD card hardware interface, including MOSI, MISO, CS, reset, MOSI enable, and MISO enable. Constant blocks were used to ensure that the SD card is enabled, along with the MOSI and MISO data lines.

Testing for the sd_controller block consisted of writing a block of data and reading it back using hardware. An ILA was used to capture the read back block and then manually inspected to make sure the block matched what was written. The SD_MYCODE block was tested by using the AXI VIP and verifying that the interface did what we expected. Then MicroBlaze was used to read and write several blocks and check that the block was not corrupted.

4.1.3 AXI Uartlite

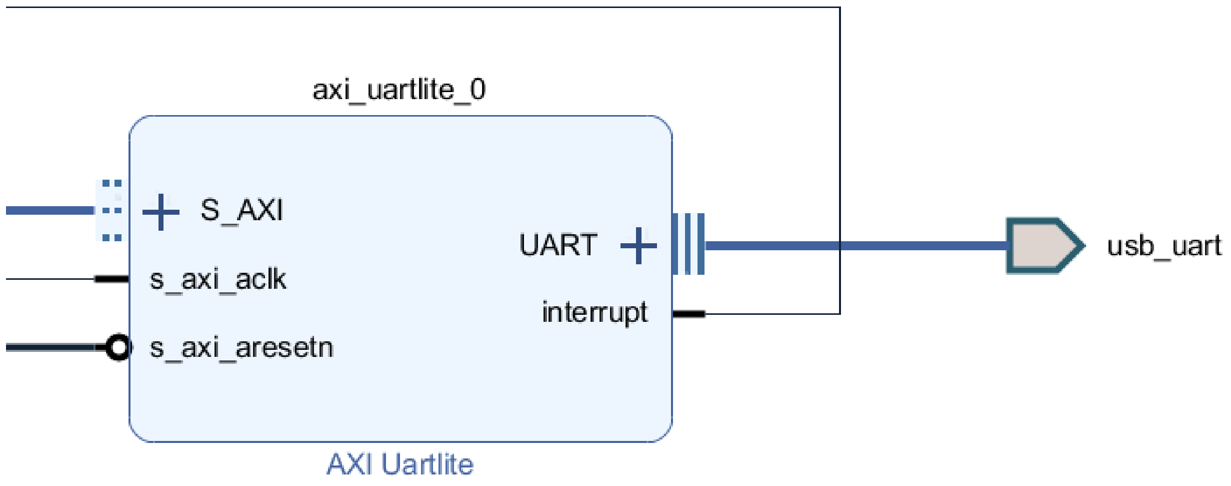

Figure 5 - AXI Uartlite block

In Figure 5, our AXI Uartlite Block is shown. We used the default AXI Uartlite IP from Xilinx and configured it to run at a baud rate of 9600, have 8 data bits, and with no parity bit. This block was used so that we could have print statements for debugging and logging in MicroBlaze. As we only used this block for logging and debugging (we only transferred small messages), a baud rate of 9600 was adequate and there was no need for us to increase the baud rate for more performance. The block sent the data from the print statements to a terminal on the PC connected to the FPGA. Testing for this block consisted of verifying that messages could be printed onto the Xilinx SDK terminal or other terminals and that the contents of the messages were correct.

4.1.4 AXI Timer

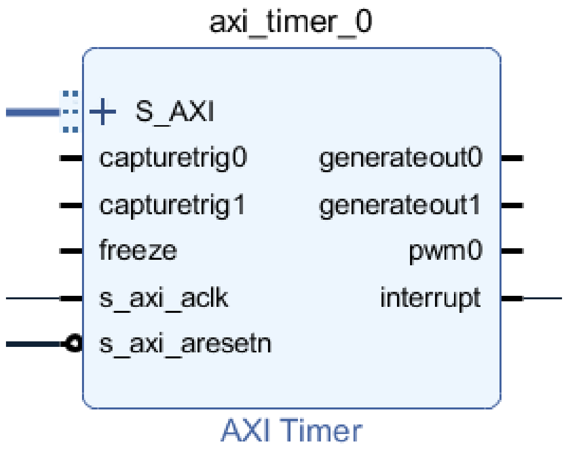

Figure 6 - AXI Timer block

In Figure 6, our AXI Timer block is shown. We used the default AXI Timer IP from Xilinx. It was configured to have a timer width of 32 bits as we expected none of our profiled sections to take longer than 42 seconds (the maximum seconds the AXI Timer block can record with 32 bits, calculated by: $(100MHz)^{-1} * \ (2^{32} - 1)$). This block was used to measure performance of various sections of code inside the MicroBlaze. It was used to profile how long it took to send an unencoded video to the FPGA from a PC, an encoded video from FPGA to FPGA, encoding and decoding time, and the time required to write the encoded video to the SD Card. Testing for this block consisted of comparing the timing measurement against another timer, such as our phone, and checking if the two were approximately equal.

4.1.5 MicroBlaze

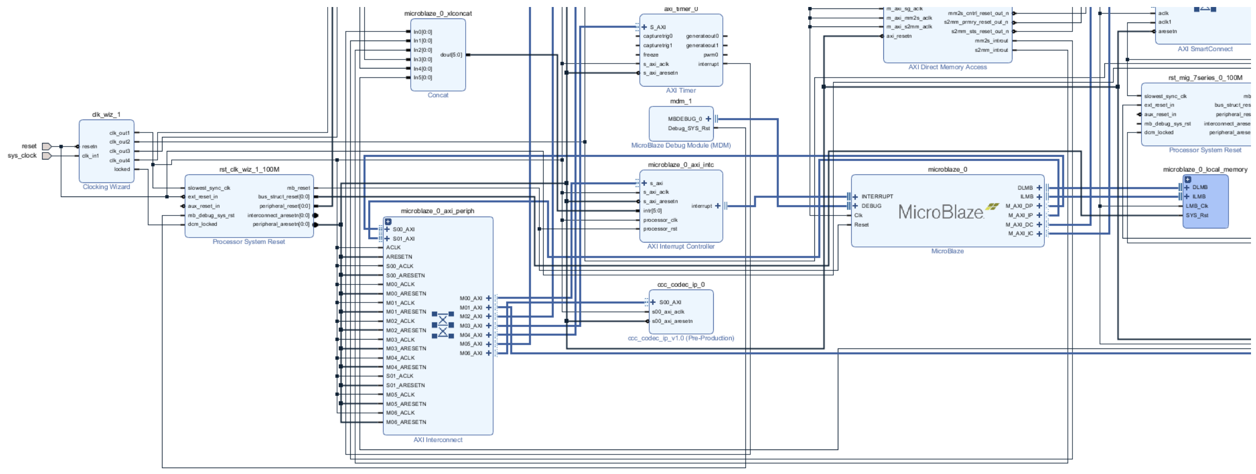

Figure 7 - MicroBlaze system with auto generated IP blocks

In Figure 7, the MicroBlaze soft processor is shown along with auto generated IPs to support our MicroBlaze configuration. The auto generated IPs are the MicroBlaze, MicroBlaze Debug Module (MDM), AXI Interrupt Controller, AXI Interconnect, Concat, Clocking Wizard, and Processor System Reset. All the IPs mentioned are from Xilinx. Our MicroBlaze configuration was created using Block Automation and enabling the AXI Peripherals and Interrupt Controller options. Everything else was left to their default values for Block Automation.

MicroBlaze was used to run our FPGA Server and FPGA Client and to interface with the CCC codec and the SD Card. All peripherals such as the AXI Uartlite, AXI Ethernet Subsystem, AXI Timer, AXI DMA, SD MYCODE (SD Card AXI Interface), and CCC codec use the AXI Interconnect and Concat blocks to communicate with the MicroBlaze. Additionally, the program running on the MicroBlaze was stored in the DDR3 memory which was accessed using the Memory Interface Generator.

4.1.6 AXI 1G/2.5G Ethernet Subsystem

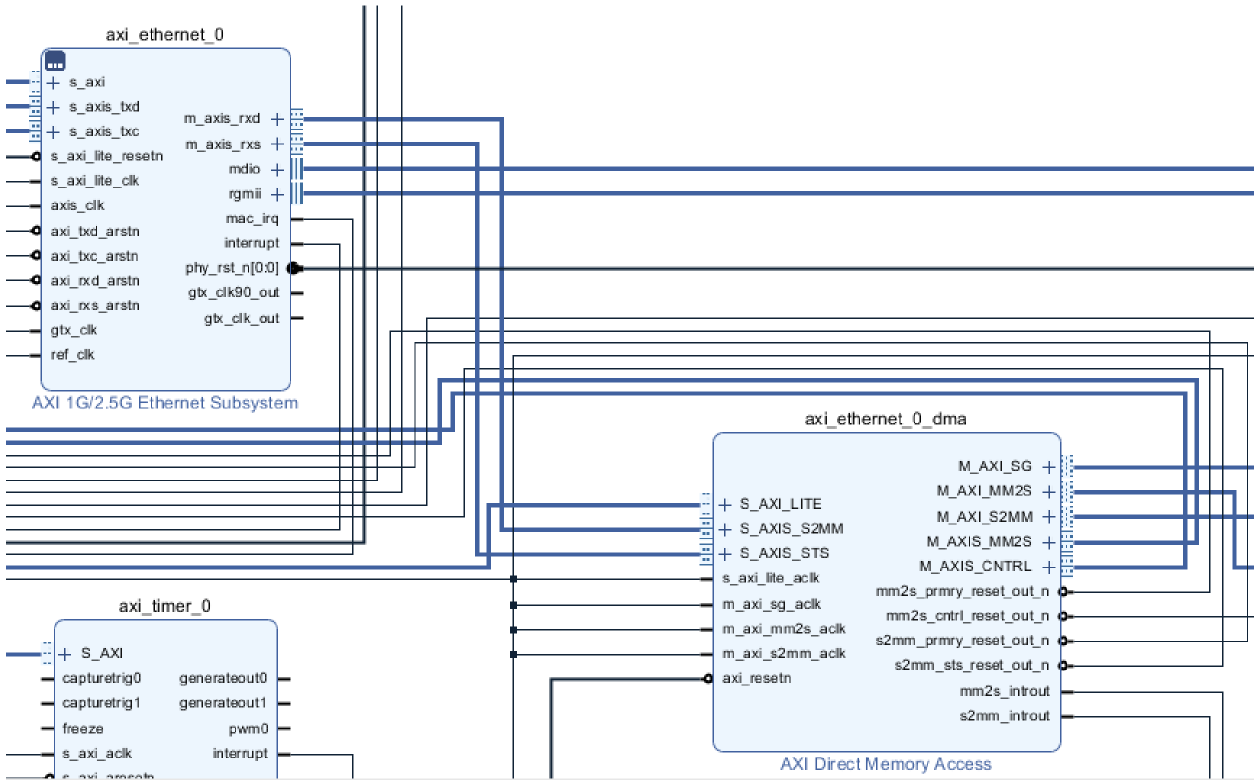

Figure 8 - Auto generated AXI Ethernet Subsystem and AXI DMA

In Figure 8, the Ethernet Subsystem is shown along with an auto generated AXI Direct Memory Access (DMA). The Ethernet Subsystem and DMA were configured using Block Automation following the configuration instructions on the tutorial provided on how to use LWIP [4]. Additionally, the Ethernet Subsystem uses 3 pins which are eth_mdio_mdc, eth_rgmii, and phy_reset_out and these pins were auto generated. The MicroBlaze, Ethernet Subsystem, and

Memory Interface Generator for the DDR3 memory all use the DMA to transfer data between each other.

The use for the Ethernet Subsystem is to send and receive video frames over FPGANet. Testing consisted of verifying that packets could be sent between two different FPGAs or from a FPGA to a PC with no corruption. This verification was done by inspecting a packet once it was received by an FPGA using a debugger.

4.1.7 Memory Interface Generator (MIG 7)

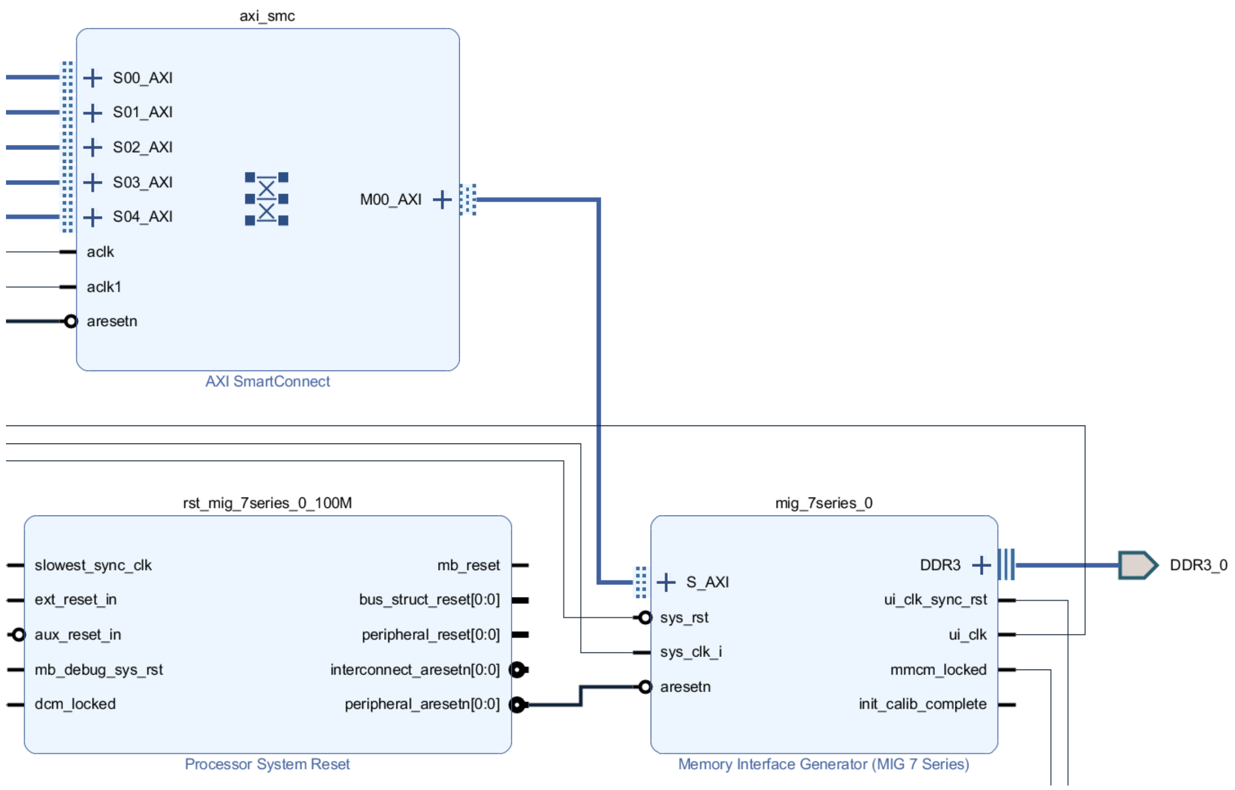

Figure 9 - Memory Interface Generator and its auto generated supporting blocks

In Figure 9, the Memory Interface Generator (MIG7) along with the auto generated AXI SmartConnect and Processor System Reset blocks are shown. The MIG7, AXI SmartConnect, and Processor System Reset were configured using Block Automation. Additionally, the MIG7 uses an additional pin, DDR3_0 as shown in Figure X. The AXI SmartConnect is used by the Ethernet Subsystem and MicroBlaze to access the MIG7.

The purpose of the MIG7 is to provide an interface to the DDR3 memory that was on our board. We required the use of external memory as the LWIP library and the video itself were too large to fit onto the memory on the FPGA board. Testing for the MIG7 block was done by storing data onto the DDR3 memory and reading it back and checking if the read back value matched what we wrote.

4.1.8 Clocking Wizard

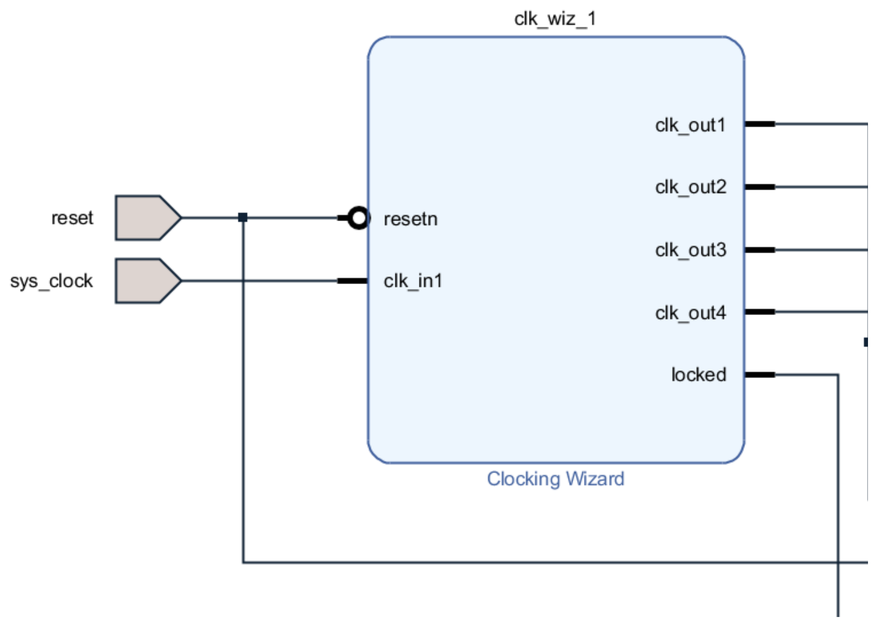

Figure 10 - Clocking Wizard used to generate clocks for the entire system

In Figure 10, the clocking wizard can be seen. The clocking wizard was auto generated when the MicroBlaze was configured. It generates a 100MHz clock that every block uses besides the sd_controller block. It was modified to generate a 25 MHz clock for the sd_controller. This was done as the clock specified for that block was 25MHz in a comment describing the sd_controller's input clock.

4.2 Software IP

All the software blocks from Section 4.2.1 - 4.2.3 and 4.2.6 - 4.2.9 were written in Python and run on a PC. The source code for Sections 4.2.1 - 4.2.3 is contained inside frame_splitter.py and Sections 4.2.6 - 4.2.9 in frame_viewer.py. The software blocks from Sections 4.2.4 - 4.2.5 were written in C and run on the MicroBlaze. The code for both sections is located inside main.c, echo.c, and xaxiemacif_physpeed.c. Sections 4.2.10 - 4.2.16 contain which libraries and drivers were used.

4.2.1 Video to Frames Splitter

This block was used to split an input video into several frames on the PC Server. This was done by opening a video using OpenCV's function VideoCapture(). Then a frame could be captured by calling read() on the opened video using OpenCV. By repeatedly calling read() more frames would be generated until enough frames were created to show the whole video. In total, this created 137 frames for our sample 6s video. This block was tested in combination with the RGB Formatter.

4.2.2 RGB Formatter

The RGB Formatter block was used to resize the frames of the video to 160x120 pixels and convert them into PPM files which use the RGB 888 format. This was implemented by using OpenCV's resize() to change the size of a frame to 160x120 pixels from the input frame size of 640x480 pixels. Then OpenCV's imwrite() was called to write the frame into a PPM file. OpenCV handled converting the frames into PPM files. Testing for both the Video to Frames Splitter block and RGB Formatter block were combined. Once a frame was created and written into a PPM file, a PPM file viewer was used to view the image and it was compared to the source video, visually.

4.2.3 Desktop TCP Server

The Desktop TCP Server block handled sending the PPM files to the FPGA TCP Server. The Desktop TCP Server block first listened for any incoming TCP connections on a TCP socket using the socket library. Once a connection was accepted, the block would send the first frame using sendall(). Then the block would wait until the FPGA TCP Server sent any message specifying that it had received the frame. Then the next frame would be sent in the same fashion until all 137 frames were sent. The data sent to the FPGA TCP Server was a fixed value so there was no need to send a packet to the FPGA TCP Server specifying the size of the data to be received. This allowed the network connection to be simplified between each PC and FPGA. Each frame was sent one at a time to allow for easier debugging.

Testing was done along with the Desktop TCP Client block. Specifically, a connection was established between the two and the video frames were transferred. Then the frames were inspected for any corruption once the entire video was transferred.

4.2.4 FPGA TCP Server

The FPGA TCP Server was primarily responsible for receiving the un-encoded video frames, encoding them, storing the encoded frames into a SD Card and then sending the encoded frames to a FPGA TCP Client. To use the FPGA TCP Server code set the macro CLIENT to 0 inside main.c and echo.c.

Frames were received from the Desktop TCP Server by first connecting to the Desktop TCP Server, waiting for an entire frame to be sent, and then sending a packet to the Desktop TCP Server requesting the next frame to be sent. This was repeated until the entire video was sent. The FPGA TCP Server knew when it had received an entire frame as each frame had a fixed length, in our case 57615 bytes, so it could just keep track of the number of bytes received. It also knew the total number of frames so from that point it was easy to keep track of which frame had been sent using the total number of bytes received so far. The un-encoded video was then stored inside a global array.

Once the video had been received it was sent into the encoder using AXI Lite. The encoder worked on a granularity of 4x4 pixels so the FPGA TCP server had to properly arrange the video data for encoding which can be seen in encode_video() in main.c. The encoded video was then stored into a separate global array.

After encoding, the video was stored on the SD Card. The encoded video was written into the SD Card one block (512 bytes) at a time. This was done by using the AXI Lite interface described in Section 4.1.2.

After completing the above steps the FPGA TCP Server waits for the FPGA TCP Client to connect before it starts sending over the encoded video frames.

There is also a timer running for how long it takes to receive the un-encoded video, encode it, and write it to the SD Card. The timer uses the AXI Timer block and its associated driver to accurately measure the performance.

Testing for the FPGA TCP Server was done one step at a time. First it was verified that packets could be received and the packet was manually inspected to check if the data matched what was expected. Then a simple system level test was performed with the Desktop TCP server sending a video to the FPGA TCP Server. Then the FPGA TCP Server sends the video to the Desktop TCP Client un-encoded. It was verified that the video was uncorrupted. Then the FPGA TCP Client was included in the system level test. The full system level TCP test was a Desktop TCP Server sending a video to the FPGA TCP Server. Then the FPGA TCP Server sends the video to the FPGA TCP Client un-encoded. Finally the FPGA TCP Server sends the video to the Desktop TCP Client. Then the frames were checked for corruption.

After the network test was successful, the encoding, SD Card, and AXI Timers were independently verified to work within the code and then added to the system level testing described above. The final system level test was as follows. The Desktop TCP Server sends a video to the FPGA TCP Server. Then the FPGA TCP Server encodes the video, writes it to the SD Card, prints out performance measurements, and sends the video to the FPGA TCP Client. Finally the FPGA TCP Server decodes the video and sends the video to the Desktop TCP Client. Then the frames were checked for corruption.

4.2.5 FPGA TCP Client

The FPGA TCP Client was primarily responsible for receiving encoded video frames from the FPGA TCP Server, decoding them, and then sending the decoded frames to the Desktop TCP Client. To use the FPGA TCP Client code set the macro CLIENT to 1 inside main.c and echo.c.

The transfer of frames functions almost exactly the same as the FPGA TCP Server. The FPGA TCP Client would connect to FPGA TCP Server first. Then a frame would be sent from the FPGA TCP Server. After receiving the whole frame and storing the frame in a global array then the FPGA TCP Client would request the next frame. The only difference compared to the FPGA TCP Server is that the frame size would change from 57615 bytes to 9615 bytes due to the encoding. As before with the FPGA TCP Server, the frame size and number of frames is constant and predefined.

After receiving the video, the FPGA TCP Client would then use the decoder to get back the original video and store that inside another global array. The decoded frame size would be 57615 bytes which was the original number of bytes in each frame. The decoder also functions on a chunk of 4x4 pixels so the FPGA TCP Client has to handle the necessary indexing for each frame and this can be seen in decode_video() in main.c.

After decoding the FPGA TCP Client waits until the Desktop TCP Client connects to it to start sending the decoded video.

The AXI Timer blocks are used to measure the performance of receiving the encoded video from the FPGA TCP Server and to measure the decoding time.

Testing for the FPGA TCP Client was similar to the testing done for the FPGA TCP Server. First the packets received by the FPGA TCP Client were inspected to make sure they were correct. Then the client was put in between the Desktop TCP Server and Desktop TCP Client and the frames were checked for corruption. Then the FPGA TCP Client was added to the full system test as described in Section 4.2.4.

The decoding was verified by comparing decoded RGB pixels to the values obtained by simulation and then decoding the video in the full system test described in Section 4.2.4.

4.2.6 Desktop TCP Client

The Desktop TCP Client was primarily responsible for receiving decoded video frames from the FPGA TCP Client and writing the frames into the frame buffer. The decode video frames were received by the Desktop TCP Client connecting to the FPGA TCP Client. Then the first frame would be sent to the Desktop TCP Client. After the full frame (57615 bytes) was received the Desktop TCP Client would request the next frame and so on until the max number of frames was reached for the video (a predefined value). Each time a full frame was received it would be written into the Frame Buffer.

4.2.7 Frame Buffer

The frame buffer was simply a collection of PPM files that filled up only when all the video frames were received by the Desktop TCP Client. It was used to ensure that the video could be played smoothly at 24 FPS. As our TCP network was too slow to send video frames fast enough to be displayed at 24 FPS we instead choose to wait till all the frames were received and then display them by reading the PPM file that was sent.

Testing for the frame buffer consisted of checking that after all the video frames were received the video would begin to play.

4.2.8 Video/GUI Controller

The Video/GUI Controller handled displaying the next frame on the GUI and receiving logging and debugging messages from UART. It opened a PPM file from the Frame Buffer by using OpenCV and calling imread(). Then it would show the image on the GUI by calling imshow() in OpenCV. After 42ms (using waitKey() from OpenCV) the next frame would be read in and shown.

Another thread was created using the _thread library and this thread used PySerial functions to connect to a serial port (the COM5 port) and display the logging and debug messages onto the Windows CMD prompt.

Testing for Video/GUI Controller was combined with the GUI & Video Display testing and the UART was verified by checking if the correct messages were printed onto the Windows CMD prompt.

4.2.9 GUI & Video Display

The GUI & Video Display was created using OpenCV. By calling namedWindow() in OpenCV a window would be created that could display frames. Then buttons were added by calling createTrackbar() and providing call back functions that would stop the next frame from being shown or start showing the next frame.

Testing was done by visually inspecting that the video could be played smoothly and started or stopped using the GUI buttons.

4.2.10 OpenCV

An image processing library. We specifically used opencv-python which contains fewer packages then the full opencv package. Requirements for the library included having numpy installed.

4.2.11 socket

A networking library. Provided by default in Python.

4.2.12 PySerial

A library for communicating to serial devices. One issue that came up is that this library can’t connect to a COM port if the Xilinx SDK terminal is also using that port.

4.2.13 _thread

Low level Library for generating thread in Python. Included by default in Python.

4.2.14 LWIP

Networking library in C that is designed to be lightweight. We had to modify one of the files (xaxiemacif_physpeed.c) in LWIP to add support for the Realtek PHY the Nexys Video Boards use. This modification can be found inside our GitHub. The issue was that LWIP didn't have any support for Realtek PHYs so we had to patch in the changes the manufacturer (Diligent) provided to our LWIP driver.

4.2.15 Uartlite Driver

Xilinx provided driver to help with using UART. Was used to initialize UART, setup interrupts, and send and receive data.

4.2.16 AXI Timer Driver

Xilinx provided driver for using the AXI Timer block. Was used to initialize the AXI Timer block, configure the timer, and start/stop the timer and receive the time taken.

5.0 Description of Design Tree

The link to our GitHub repository and a video showcasing our project is provided here:
https://github.com/KanverB/G7_VidAccel-A_FPGA_Accelerated_Video_Streaming_Service

A copy of the README is also provided below.

G7_VidAccel-A_FPGA_Accelerated_Video_Streaming_Service

Video Demo

Below is a video showcasing our project.

https://user-images.githubusercontent.com/47866804/114767729-00136d00-9d36-11eb-8a88-864875088e79.mp4

Description

We built a video streaming platform, essentially something similar to Netflix but much smaller in scope. For our project, we used FPGAs to accelerate the encoding and decoding part of video streaming since this takes a significant portion of time on a CPU.

Our project uses a Desktop TCP Server to take in an input video and split it into many RGB 888 frames. Those frames are sent to a FPGA TCP Server which encodes the frames using a color cell compression hardware and stores them on a SD Card. Then a FPGA TCP Client would request the encoded frames and decode them using hardware. A Desktop TCP Client would then request the decoded frames from the FPGA TCP Client and display them on a GUI for viewing.

How to use

1. Change the host IP addr in frame_splitter.py to match your PC's IP. Modify the file path for where the source video is located in frame_splitter.py. Run frame_splitter.py. This step sets this PC as the Desktop TCP Server.

2. Create the Vivado project and generate the bitstream. Then open the Xilinx SDK and create a new project based on the lwIP Echo Server template. Replace the following files main.c, echo.c, xaxiemacif_physpeed.c (if using the Nexys Video Board), and lwipopts.h (or modify the BSP settings to match this file). Set the macro CLIENT to 0 in main.c and echo.c and change the DEST_IP4_ADDR to the addr used in the Desktop TCP Server. Also, change the mac_ethernet_address in main.c to match your board and the src IPV4_ADDR to match your board. Start the program. This step sets this FPGA as the FPGA TCP Server.

3. Then using another FPGA board follow the same steps as 2 but using the new board's IPV4 addr and MAC addr and setting the DEST_IP4_ADDR to the addr used by the FPGA TCP Server. Set the marco CLIENT to 1 in main.c and echo.c. This sets the FPGA as the FPGA TCP Client.

4. Change the host IP addr in frame_splitter.py to match the FPGA TCP Client's IP. Make sure to specify the directory that the ppm files should be written into in frame_viewer.py. This step sets this PC as the Desktop TCP Server. Then after the video has been transferred it should play on a GUI! The start and stop buttons can be clicked to start or stop the video. Debugging and logging messages are printed onto the terminal while frames are being sent from the FPGA TCP Client.

Repository Structure

- root
    - docs/: Directory holding all our documentation.
        - Final Demo Presentation: Our slides that were used in the final presentation of our project.
        - Final Report: A report providing an overview of our project, the outcome, our schedule, a description of the IP blocks, and tips and tricks.
        - Video: A video demo of our project.
    - Example_Video/: Directory containing our example source video to send using our project.
        - IMG_3138.mp4: The video we encoded and decoded. A very cute cat.
    - LWIP_Modifications/: Changes we made to LWIP driver files.
        - xaxiemacif_physpeed.c: Modifications done to the default LWIP xaxiemacif_physpeed.c file. Added support for Realtek Ethernet PHYs.
    - PC_GUI_Server_Client/: The code for our Desktop TCP Server and Desktop TCP Client. Also contains a GUI.
        - frame_splitter.py: Code for Desktop TCP Server. Also, splits a video into RGB 888 frames.
        - frame_viewer.py: Code for Desktop TCP Client. Also, displays the decoded frames on a GUI and prints to the terminal debugging and logging messages from the FPGA TCP Client.
    - src/:
        - ccc_codec_ip_1.0/: Directory containing src code and testbenches for our encoder and decoder.
            - hdl/: Directory containing our src verilog files.
                - ccc_codec_ip_v1_0.v: AXI Lite interface src code wrapper.
                - ccc_codec_ip_v1_0_S00_AXI.v: AXI Lite interface src code.
                - ccc_decoder.sv: Decoder src code.
                - ccc_decoder_4x4.sv: Decoder src code that generates multiple decoders.

            - ccc_encoder.sv: Encoder src code.
            - ccc_encoder_4x4.sv: Encoder src code that generates multiple encoders.
        - MicroBlaze_test/: Directory containing our MicroBlaze testing code.
            - ccc_codec_test_*.c: 5 test files that each run a different test on the encoder and decoder.
        - testbench/: Directory containing our testbench code.
            - ccc_codec_axi_vip_tb.sv: AXI VIP testbench src code.
            - ccc_codec_tb.sv: Src code for codec testbench.
            - ece342_vga_bmp.sv: testbench to show output as a bitmap.
            - tb_frames\: Directory containing a variety of test images in .png and .ppm format.
                - *.ppm and *.png images: A variety of test images used in the testbench for the encoder and decoder.
    - constraints/: Directory containing constraints for our boards SD Card pins.
        - SDpin.xdc: Constraints file containing our SD Card pin mappings for the Nexys Video Board.
    - MicroBlaze_Code/: Directory containing src code for FPGA TCP Server and FPGA TCP Client.
        - echo.c: The src code for our FPGA TCP Server/Client. Used to send the frames to anyone who connects. Messages or prints are logged using UART onto a terminal.
        - lwipopts.h.c: The src code for the parameters we choose for the LWIP driver code. These parameters can be modified using the Modify BSP Settings button in the system.mss file using Xilinx SDK.
        - main.c: The src code for our FPGA TCP Server/Client. Used to receive frames. Also encodes/decodes the frames and writes them to the SD Card. Messages or prints are logged using UART onto a terminal.
        - platform.c: The src code for adding interrupts for UART into MicroBlaze.
    - SD_MYCODE_1.0/: Directory containing src code and test drivers for our sd_controller interface. MircoBlaze would communicate with the SD Card using this interface.
        - hdl/: Directory containing our src verilog files.
            - SD_MYCODE_v1_0.v: AXI Lite interface src code wrapper.
            - SD_MYCODE_v1_0_S00_AXI.v: AXI Lite interface src code.
        - MicroBlaze_test/: Directory containing our MicroBlaze testing code.

            - sdcarddriver.c: Test file that runs on MicroBlaze that writes and reads blocks to the SD Card.
    - SD_SPI/: Directory containing src code of our modifications to the SD Card SPI interface we used from MIT.
        - hdl/: Directory containing our src verilog files.
            - SD_SPI.v: Src code for the modifications we made to the SD Card SPI controller from MIT.

Authors

Tianyi Zhang
Kanver Bhandal
Bruno Almeida

Acknowledgements

Professor Chow - For being an amazing professor who taught us many useful concepts that we used throughout our project.

Daniel Rozhko - Our TA who provided us with lots of helpful feedback and helped answer any questions we had.

MIT - For their open source SD Card Controller SPI IP. The original SD Card SPI code can be found here: http://web.mit.edu/6.111/www/f2015/tools/sd_controller.v

Group 1 - Hyperspectral Image Compression and Decompression on FPGAs - For providing us with help on what modifications needed to be done to the SD Card Controller IP and for sending us the source of the IP they used for the SD Card.

6.0 Tips and Tricks

Try to use a version control system if possible. We were each able to set up an installation of Portable Git for Windows in our W drives and connect it to a GitHub repository. This makes it easier to track your RTL files and code and ensure everyone has the latest versions (especially important when integrating hardware and software).

Develop a strategy early for how you are going to manage your Vivado project(s). It is difficult to manage them, so make sure you have a plan. A couple of possible options include:

- Everyone creates their own project and recreates the block diagram themself
- Manually send (or commit to version control) the Vivado project (could remove large folders such as caches)
- Export a TCL script to recreate the project and send (or commit) the script

Try to simulate all of your hardware blocks to some degree before integrating them with a MicroBlaze. Even a simple simulation can catch many common bugs quickly, e.g. not setting register values upon reset.

The AXI VIP is very useful for debugging AXI IP cores in simulation. It takes some work to set up a separate Vivado project and block diagram for a simulation, but it saved us several hours in net debugging time compared to going straight to integrating with a MicroBlaze and testing on hardware.

Test and simulate individual blocks before verifying the top level design. Start with simple test cases.

ILAs are very useful for debugging hardware when stepping through a MicroBlaze program. You can just set up ILAs to trigger, step one line in the MicroBlaze program, and see the effects (in hardware signals) when that line of code is executed.

Don’t worry about performance/speed as a primary factor of success, just get something functionally working and correct first.

Be very careful about trusting Vivado reports after running synthesis and implementation. Make sure you actually test your bitstream on hardware and ensure it is functionally correct before trusting the resource and timing reports. As in our case, your design may have bugs in the logic that cause Vivado to optimize away massive amounts of logic, reporting that it passed timing and/or fit in the FPGA resources when in fact it does not once bugs are fixed and it is functionally correct.

It is very useful to prepare a project descoping plan in advance, understanding which functionality is core to something working, and which are extra features that can be cut if time and resources are limited.

References

[1] Forse, N., 2017. Finding Acceleration in a Video-Centric World. [online] Available at: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01268-finding-acceleration-in-a-video-centric-world.pdf> [Accessed 30 January 2021].

[2] "sd_controller", Web.mit.edu. [Online]. Available: http://web.mit.edu/6.111/www/f2015/tools/sd_controller.v. [Accessed: 14- Apr- 2021].

[3] "NetPBM Viewer", Paulcuth.me.uk. [Online]. Available: http://paulcuth.me.uk/netpbm-viewer/. [Accessed: 14- Apr- 2021].

[4] S. K, "Nexys Video - Getting Started with Microblaze Servers", Digilentinc. [Online]. Available: https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-video-getting-started-with-microblaze-servers/start. [Accessed: 14- Apr- 2021].