FPGA Based: Surveillance System

ECE 532 Group 1 Final Report

Jia Yuan Chen
Han Jie Qiu
Xinran Rui

Table of Content

1.0 Overview

1.1 Introduction and Motivation

The rapid developments in network technologies such as the emerging 5G networks have made bandwidth demanding applications such as video streaming much more accessible to the public. As part of the future smart home application, video surveillance systems allow the users to monitor any point of interest at a remote destination. This convenience allows users to keep a peaceful mindset without worrying about events such as burglary.

The goal of this project is to utilize the power efficient field programmable gate arrays in developing the next generation home surveillance system with extended functionalities. In addition to network video streaming of the camera feed, the project further aims to offer an automatic motion detection feature that would notify the user in the event of break-ins. Once a motion is detected the system would record a few seconds of the video feed, notify the user through instant notification and allow the video to be replayed. The existing surveillance camera system may have more extensive features such as data encryption, but our design focuses on the lower power consumption offered through the FPGA platform. As the surveillance systems are often kept running for a long period of time, the power consumption could quickly accumulate to a very large number. Having the system implemented in the power efficient FPGA platform could help to reduce power consumption. Furthermore, the FPGA platform also allows future reconfiguration to extend support for data encryption and network protocol changes.

This report would present the system level design followed by detailed module level descriptions and design choices. We would also comment on the quality of our outcomes as well as the verification methods during integration. Finally, we conclude with various tips and tricks in using the Vivado design tool from Xilinx.

1.2 Project Goal

The goal of this project is to develop a network surveillance system with video streaming and motion detection capabilities.

1.3 Resource Requirements

Table 1. below shows the list of required resources that the group used during the development of this project.

1. Xilinx Vivado	2. Nexys Video Board	3. OV 5640 Camera
4. HDMI Output & Monitor	5. Xilinx TEMAC IP License	6. Xilinx Video Mixer License

Table 1 Resource Requirements

Note that the TEMAC IP license is required to enable the network capabilities on the Nexys Video FPGA. This license could be acquired from Xilinx with 120 Days of node-locked evaluation license.

1.4 System-level Block Diagram

This section provides the system-level block description that highlights the major components with the proposed design. The system consists of three major components: Camera Capture and Playback Subsystem, Motion Detection Subsystem and Network Subsystem. Figure 1 below shows the system diagram and the connections between the three major components.

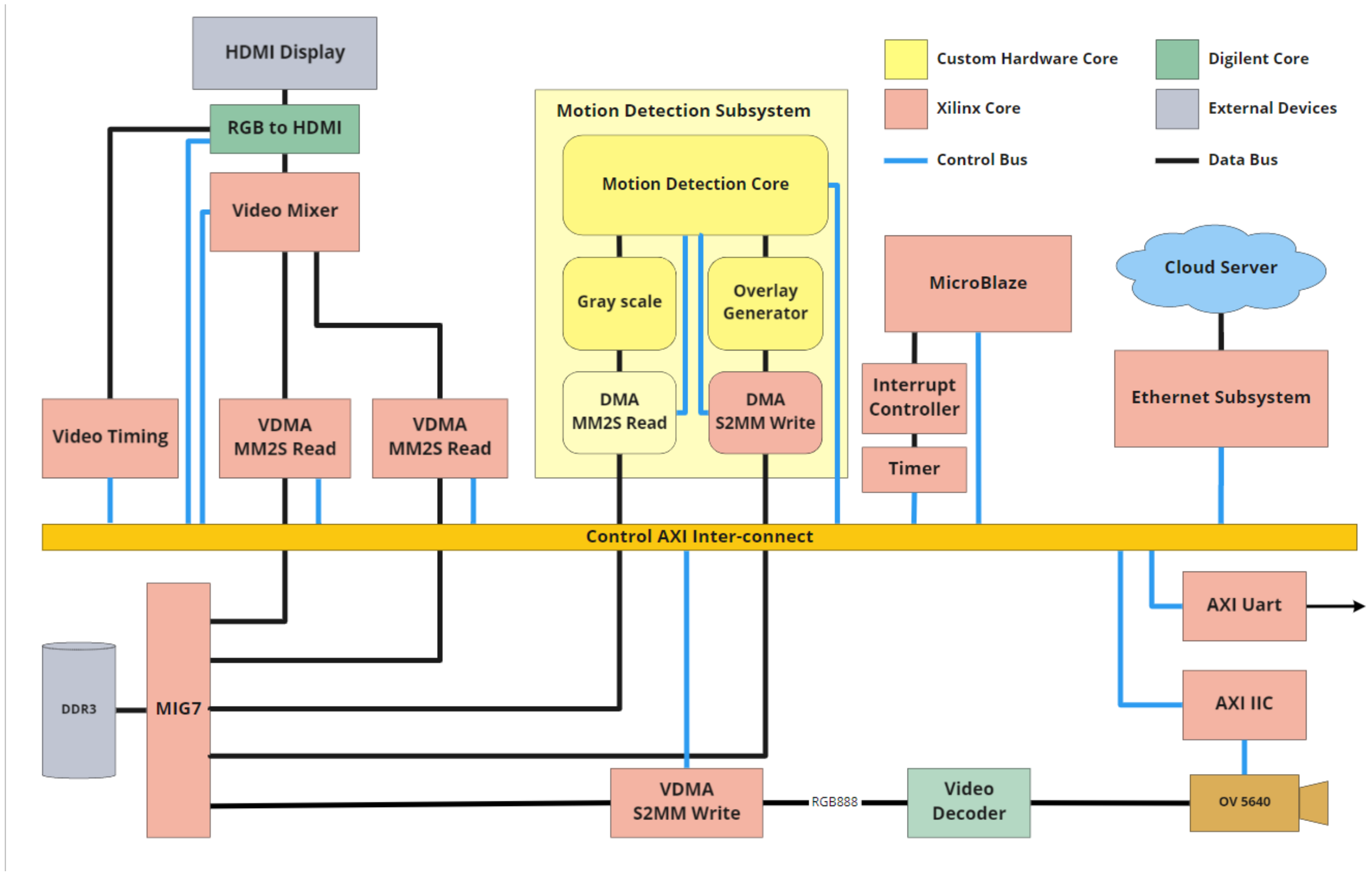


Figure 1. System-level Overview

Camera Capture and Playback Subsystem

The system utilizes an OV5640 HD PMOD camera to capture the live video data. For debugging purpose and demonstration of the motion detection function, an HDMI video output was added to the pipeline to allow viewing of the camera feed on the local monitor.

Motion Detection Subsystem

With the video data captured by the camera and saved to the DDR, the motion detection subsystem reads and performs the motion detection algorithm in determining if there exist new movements inside the camera feed. If a motion is detected, it would notify the MicroBlaze processor and pass the message up the network stack to the remote server.

Network Subsystem

The MicroBlaze processor and the Ethernet subsystem together defined the network subsystem for our design. It allows the client application running on the MicroBlaze processor to communicate with the remote server and send information about the system status, live camera feed and motion detected results. The MicroBlaze is used to construct the data packets and send the packets through either TCP or UDP protocols depending on the use case.

1.5 Intellectual Properties Used

Table 2 below shows the list of primary IPs used in this system and how they are used.

Xilinx	VDMA	The video direct memory access IP was used extensively throughout the framework in controlling the video data flow. It was used to capture the video signal from the camera to the memory and read from memory to the RGBtoDVI core for HDMI.
	Video Timing Controller	The video timing controller was used to generate the synchronization signals for the video outputs. It works in parallel with the VDMA to serve as inputs to the RGBtoDVI core. It was configured to operate at 640x480.
	Memory Interface Generator	The MIG IP was used to interact with the DDR memory present on the Nexys Video board. The DDR memory was used as the primary storage as the BRAM was not big enough to hold a whole frame of video input. (640x480x3 = 230400 bytes)
	AXI IIC	The IIC IP was used to communicate with the OV5640 camera module through the PMOD connectors and program the camera to obtain the correct resolution, frame rate, and output format.
	AXI 1G/2.5G Ethernet Subsystem	The ethernet subsystem was used to provide the system with the network capability. Through the LWIP APIs, the system could send both TCP and UDP packets through the ethernet connection to the remote server.
	Video Mixer	The video mixer IP was used to enable the alpha-blending feature of the framework. It allows the system to overlay and highlight the region where the motion occurred.
	MicroBlaze	The MicroBlaze processor is the soft processor that is used in this system. It was used to perform the necessary initialization of various IPs as well as constructing the network packets for the ethernet subsystem for transmission.
Digilent FPGA	RGBtoDVI	The RGBtoDVI IP from Digilent was used to convert the RGB signals from the system to the DVI HDMI output.
	Dynamic Clock Generator	The dynamic clock generator generates the necessary serialization clocking source for the RGBtoDVI core. Through this core, the system could output video frames at various resolutions but now the system only uses the 640x480 VGA resolution.
Custom Hardware	Motion Detection Subsystem	The motion detection subsystem is the custom hardware core that performs the motion detection algorithm and notifies the MicroBlaze when motion is captured.

Table 2 Primary IPs used

2.0 Outcome

This section goes over the verification methods that the team used to validate the functional correctness of the system as well as the quality of the results obtained from the final implementation.

2.1 Verification Methods

The team designed the project using the bottom up approach by building and testing the modules incrementally before the final integration. Major modules such as the motion detection subsystem were implemented and tested against a golden software implementation to validate the quality of results.

Below are the major methods that the team utilized during the verification stage:

Hardware Simulations

The team used the hardware simulator and the VIP IP offered by Vivado extensively. Using the VIP IP allows the team to test the framework without the need of the MicroBlaze processor which greatly improved the simulation speed. Furthermore, the simple AXI Lite interface offered by the VIP IP also allows the team to quickly validate if each read/write transaction was successful. Together with the $display and $finish directives, the team was able to eliminate multiple bugs and issues in the submodules before the final integration.

However, one of the drawbacks in using that VIP IP was that it requires the system to be synthesized before each run of the simulation which usually takes a while. To allow a faster turnaround, the team moved away from the VIP IP and used custom Verilog *tasks* to simulate the AXI Lite handshake which does not require synthesis and greatly improve the turnaround time in identifying and resolving issues. The custom Verilog *tasks* will be shared as one of the community contributions from this group to the class.

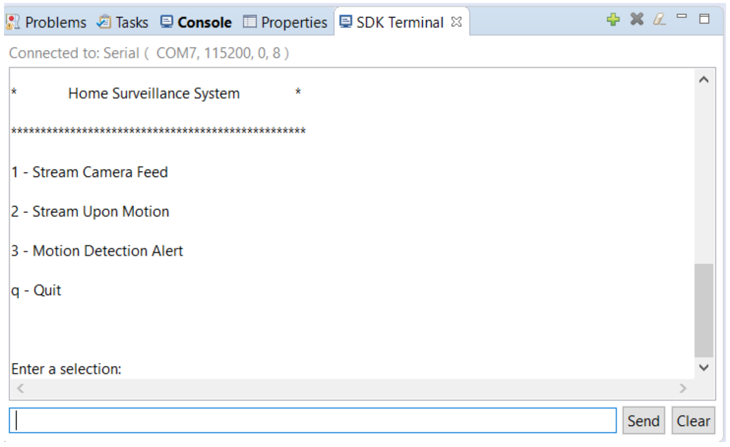
Integrated Logic Analyzer (ILA)

While the behavior simulator solved most of the logical issues during the design of each submodule, the team struggled during the integration process. As the most common problems exist during integration are often related to interface handshakes, the use of the ILA for runtime probing allows the group to quickly find deadlocks in handshakes and made integration possible.

2.2 Results

The group was able to successfully deliver the baseline implementation of the network video streaming framework with motion detection trigger support. Furthermore, a local HDMI output was also successfully implemented to allow video output on a monitor for inspection and debugging purposes.

Figures 2 – 5 below shows the sample output of the completed framework.



```
Connected to: Serial ( COM7, 115200, 0, 8 )
*         Home Surveillance System          *
**************************************************
1 - Stream Camera Feed
2 - Stream Upon Motion
3 - Motion Detection Alert
q - Quit

Enter a selection:
```

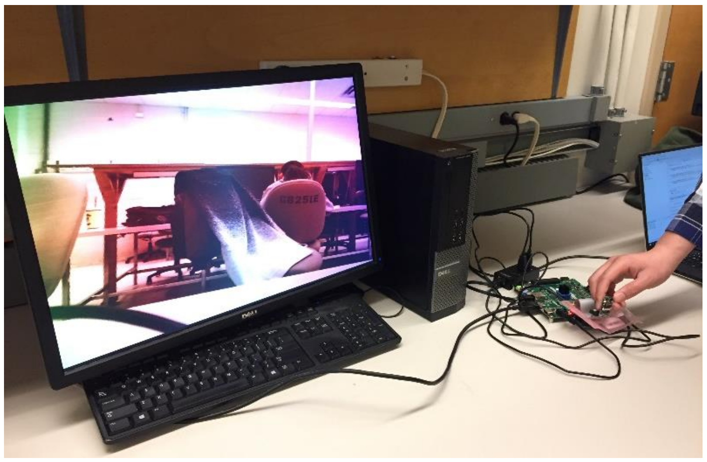
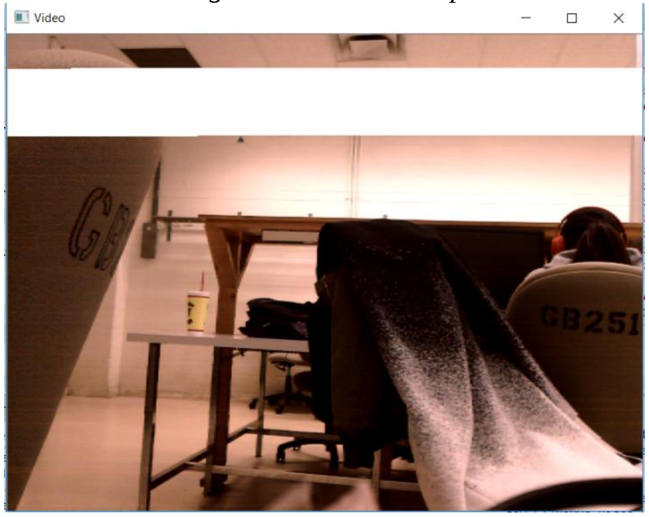
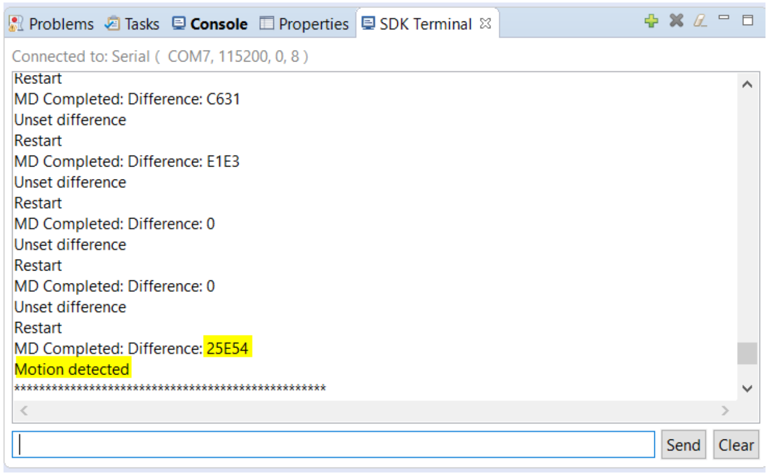


Figure 2 SDK Menu Options

Figure 3 HDMI Output of Camera Feed





```
Connected to: Serial ( COM7, 115200, 0, 8 )
Restart
MD Completed: Difference: C631
Unset difference
Restart
MD Completed: Difference: E1E3
Unset difference
Restart
MD Completed: Difference: 0
Unset difference
Restart
MD Completed: Difference: 0
Unset difference
Restart
MD Completed: Difference: 25E54
Motion detected
**************************************************
```

Figure 4 Sample Video Streamed to Host

Figure 5 Motion Detection Output

In addition to the above functionalities, the team also wanted to support a motion detection overlay to the video output that highlights the region of where motions were detected. However, while this function was successfully validated in the behavior simulation, the team experienced issues during the deployment onto the Nexys Video board.

2.3 Motion Detection Overlay Design

Figure 6 below shows the desired goal of the overlay feature. As shown, the group wish to highlight the region of where the motion occurred in the video to better describe the detected motion trigger.

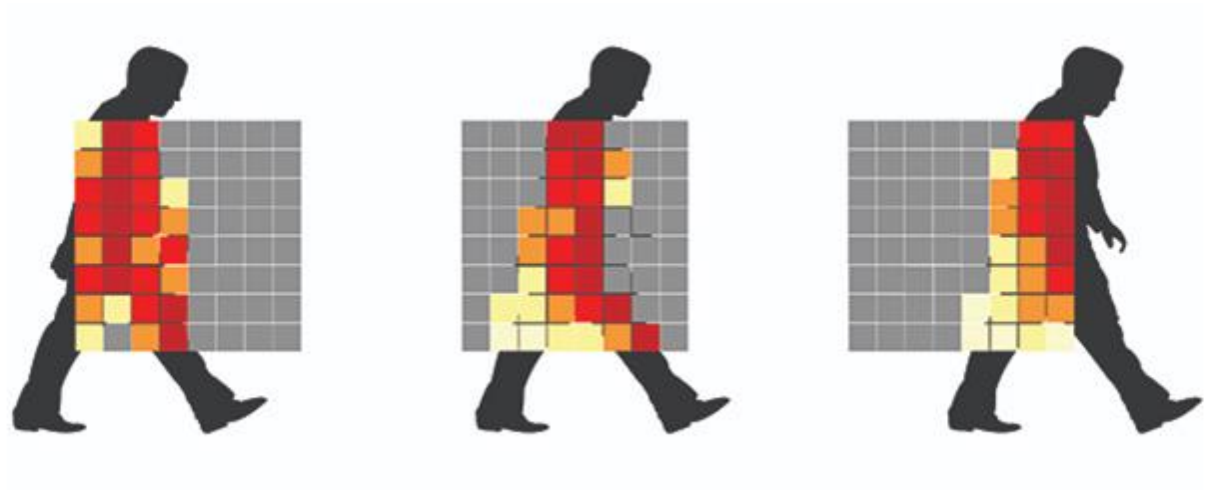


Figure 6 Expected Output of the Overlay Function

As shown in figure 6, we divided the video frame into multiple small squares and perform motion detection in each square. The motion detection algorithm that we used was calculating the absolute difference between two frames. We first convert the frame from RGB into grayscale and calculate the absolute difference of the grayscale values of two pixels at the same location in different time frames and sum the difference across all pixels. When the total difference exceeds a certain threshold, motion is detected. Pixels with the greatest differences are then considered to be the location of motion and should be highlighted.

One of the challenges that we encounter was that the video data was stored in RGB888 – 24bit format while the data mover used the send data to the motion detection core can only operate at 32, 64 or 128 bits. To ensure high-speed execution, instead of padding the pixel data with an extra blank alpha channel to make it 32-bit wide, the team had an alternative design. With the DMA moving 32-bit at each cycle into the motion detection core, we waited until we capture 3x32bit worth of data before processing. 3x32-bit = 3 words = 4 pixels. This means that every 3 cycles we will operate directly on these 4 pixels. This method allows us to avoid using the extra alpha channel which saved and helped with the network throughput after. However, because of this, it becomes more difficult when we need to calculate the square-wise frame difference because we must pick values that would align with both the frame resolution, the pixel width as well as the 3-word-4-pixel processing. We determined to use 15x20 word size in the end because it aligns with our memory access.

With the square differences calculated, we wanted to use the DMA write channel to write the square outputs that contain the motion intensity into a different memory region. We then use the Video Mixer IP to perform an alpha blending between the camera feed video source and the square motion intensity on the same output stream.

We were able to perform the above executions and validate the correctness through the behavior simulation. However, we were not able to deploy. The problem that we encounter exists in the DMA core that the motion detection subsystem utilizes. To ensure that we operate at a high throughput, the group used the DMA core extensively because of its S2MM and MM2S capabilities. While the MM2S read channel operates as expected and was able to repeatedly read data from the same memory region, the S2MM write channel experience major challenges. It was discovered that the first write operation would always be successful while the second repeat write would hang as the tready signal from the DMA would remain low forever. The team had tried by resetting the DMA after each read and write cycle, but the problem persists.

The solution to this problem is that the team could use the scatter-gather mode of the DMA core instead of the current Simple AXI mode. The team could create an FSM that generates the CMD signals that contain the write command and monitor the STS status signal coming back from the DMA. This method is proved to work and generally the correct approach used in the industry. However, the team did not have enough to write this FSM as it would require an extensive amount of research and investment of time to have this working. On the other hand, the team did manage to have the Video Mixer IP working which enables the alpha blending capability of multiple video streams. A tutorial of how to use the Video Mixer IP would be made available for future students of this course. This Video Mixer could allow implementations such as games much easier by maintaining different graphical assets on different frames and only perform the merging in the last output stage.

What would you do differently if you could start over or continue with this project?

1. **Operate the DMA in the scatter-gather mode to allow cyclic read and write capabilities.**
   As discussed above, the current implementation uses the simple AXI method which has issues in performing cyclic writing operations.

2. **A better algorithm for Motion Detection**
   While the current absolute frame different algorithm shows to generate very nice results, we could investigate into other algorithms that can help to reduce false positives.

3. **Implement a caching system to increase the throughput for the network transfer**
   The current implementation could achieve 1 FPS @ 640x480 RGB or 4 FPS @ 320x240 RGB. It was found that this result could be better if we allow the ethernet subsystem to access the memory more efficiently. Since the current system places everything in the DDR, having the caching system would remove stress on this common bus and achieve higher throughput.

3.0 Project Schedule

Table 3 below shows the comparison between the original project schedule and the actual progress that the group made. Modifications and justifications for major changes can also be found below.

I.D.	Original	Actual
1.0	**DMA Test Bench and Simulation** A DMA system with S2MM and MM2S capability	**Complete as expected** The group got familiar with the DMA core and was able to program the DMA core to perform the desired tasks.
2.0	**Camera Feed** Configure the camera to provide RGB stream data to the system **TFT Controller Output to VGA Monitor** Configure the TFT Controller to take RGB stream data and output the frame via VGA to a monitor **Network File Transfer** Evaluate the throughput of the file transfer to determine the target resolution and framerate used for network video streaming	**Complete as expected** Each of these three modules was completed on separate test benches and was ready for system integration.
3.0	**Basic Motion Detection Algorithm** Combining with the DMA Test Bench, construct a motion detection hardware implementation in simulations	**Complete as expected** **Addition:** Switched from Nexys DDR 4 to Nexys Video for the larger resource. VGA core was removed and replaced with HDMI core and further modifications were made to run the OV 5640 camera on the new board.
4.0	**Enhance Motion Detection Algorithm** Enhance and fine-tune the motion detection algorithm to enable better accuracy **Other Supporting Video Processing Cores** Supplementary video processing cores such as gray-scale conversion, hue-shift, and edge-detection that may help with motion detection	**Complete as expected** The group was able to deploy the entire system on the Nexys Video board and was able to fine-tune the motion detection custom core to achieve good results. However, we had issues with timing, so we investigated into resolving these problems.
5.0	**Server-Side Application** A server-side application that is capable to communicate with the client application running on the MicroBlaze and decode the receiving packet for display **Client-Side Application** A client-side application that runs on the MicroBlaze processor that is capable of image packet construction and encapsulation for network transfer	**Complete as expected** Basic server-side and client-side applications were implemented, and pictures were able to be sent from the FPGA client to the server running on the host computer. However, the team discovered that the throughput was horrible with TCP and live video streaming could be unachievable.
6.0	**Mechanical Component for Camera Angle Adjustments** A mechanical system that can adjust the camera angles in two dimensions. The adjustments are expected to be done with motors and power-width modulation.	**Deleted.** We deleted the mechanical components of our system and focused on achieving video streaming through the network. We researched into network protocols and found that the UDP protocol could achieve better throughput, so we modified our software to use UDP for data transfer.

7.0	**Application Extension** Extend the functionalities of the applications in supporting features such as motion sensitivity detection, high framerate grayscale streaming.	**Complete as expected** As an extension to the system, we decided to do a motion detection overlay as described in section 2.0. We were able to successfully simulate the behaviors but had issues when deploying to the board.

4.0 Description of the Blocks

In this section, we describe in details of the major IPs used in this system.

4.1 Custom Hardware Core – Motion Detection

The custom hardware core that the group implemented for this project is the motion detection core used to determine if there exists a new motion inside the video feed. Having this core allowed the system to only notify the user when a new event occurred in the surveillance system, helping the user to save time and effort.
The group started the design of this custom core from a golden software implementation. Through research, trials, and errors, the group discovered an effective algorithm that has very high correctness in determining motions in the video feed. The algorithm takes two video frames from two different time frame and converts both to grayscale. It then determines the absolute difference in grayscale intensity across all pixels and accumulates these differences over the whole frame. If a new object enters the frame, the color intensity of those pixels would change dramatically and cause a greater sum of absolute difference. Through iterative tuning, the group found a suitable lowpass threshold that was able to filter the noise from the camera when the scenery is static.

With the algorithm tested and verified on the software, the group built a dedicated hardware platform to test the motion detection core independent of the rest of the system. Figure 7 below shows the block design of the test bench that we built. To ensure that the core correctly calculates the frame difference, the group wrote custom Python scripts to generate test patterns in COE format and initialize the BRAMs in the block design to be ROMs. To allow line rate processing, the group wanted to use the AXI Stream interface which provides the greater throughput. However, since the video frames are saved in memory (DDR in the system and BRAM in the testbench), we need to access the memory to fetch the pixel information. If we simply use an AXI Lite interface, it would take multiple cycles to get a single word length data which generates a very high overhead. Alternatively, the group decided to use the AXI Lite interface to control the Xilinx AXI DMA and performs an MM2S read operation that fetches data from the memory and pipe to the core in AXI Stream interface. This method allows the core to operate at almost one word per cycle. With the target operating frequency of 100Mhz, the core at most can process 100M word per section. With the camera programmed to output at 640x480x3@60Hz, it generates 6912000 bytes = 1728000 words of data per second. This means that the core could operate much faster than the data source. By consulting the user manual of the AXI DMA IP, the team was able to operate the DMA MM2S channel in a cyclic fashion the pipes the core with entire frame worth of data with tlast indicating the frame boundary.

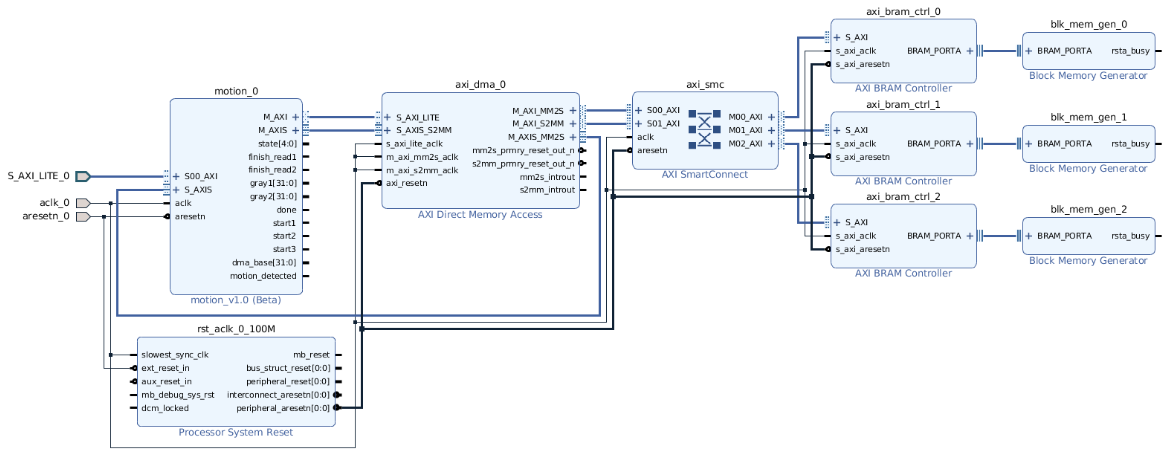


Figure 7. Motion Detection Subsystem Test Bench

With the above implementation correctly implemented and integrated into the system, the group was able to achieve the proposed design. The system could determine new motions in the video feed and notify the MicroBlaze which then propagate the message to the remote server.

However, to further extend the capability of the system, the group also wanted to create a motion detection overlay on top of the video source. As described in section 2.2, the group divided each frame into smaller grids and tries to calculate the absolute gray scale difference inside each grid. We then normalize the difference intensity and create a frame of mosaic-like grids with darker grids representing regions of high motion intensity and lighter grids representing static scenery. We write this mosaic layer into a separate memory region through the S2MM channel of the AXI DMA IP. To merge the mosaic layer with the video source, we utilized the Xilinx Video Mixer IP which offers alpha blending capability between multiple video streams. While the group was able to get the Video Mixer IP deployed on the board, the group encounter difficulties in operating the S2MM channel in cyclic mode.

Through a similar programming sequence, the group was able to perform the first S2MM write operation through the AXI DMA core. However, the group could not reset the DMA correctly in operating the S2MM operation again. The tready signal from the DMA would remain low forever which mean that it could not accept data from the stream. After researching online in Xilinx forums, we discovered that we had to wait for extra cycles after the reset even when the tready signal is high. Therefore, we added extra delays between the reset and the next operation. However, the problem persists, and we could not find a solution to fixing this issue on time. Alternatively, we could operate the DMA in Scatter-Gather mode as described in section 2.2 with a separate FSM to maintain the CMD and STS signals to achieve the cyclic operation correctly.

4.2 Video Direct Memory Access

The system used many direct memory accessors to offload the data moving task away from the MicroBlaze in achieving high throughput executions. We used a video DMA variant as it better handles the pixel RGB888 data type as well as operating the memory accesses in non-word-align 24-bit accesses. The VDAM was used to capture the video source from the OV5640 into the memory buffer and playback the video source from memory to the RGBtoDVI core for HDMI output. The simpler DMA version was used as data movers for the custom motion detection core to ensure line rate execution.

4.3 Xilinx Video Mixer

The Xilinx video mixer IP was used to perform alpha blending between two or more video sources and output as a single stream. It was intended to be used to achieve the overlay capability of the motion detection algorithm but because the group encounter issues with the S2MM channel of the DMA, the video mixer was not properly used in the system. However, the group thinks that this video mixer could be very useful for graphically demanding applications such as object detection, video processing and video gaming on the FPGA. Therefore, as a community contribution, the group would create a tutorial in explaining the use of this core to the class.

4.4 Xilinx Ethernet Subsystem

The ethernet subsystem from Xilinx is the heart of the IoT concept for this project. It allows our surveillance system to relate to the network and sent data across the network to a remote user. The group used the MicroBlaze processor and the LWIP APIs to construct and send network packets in both TCP and UDP protocols to a remote server.

The TCP network protocol provides desirable characteristics such as no-packet-loss, in order transfer and acknowledgment feedback. However, the group discovered that these characteristics came with a heavy penalty on network throughput. With the resolution of the 640x480x3 worth of data at every frame, the TCP transfer struggles to achieve “live feed” throughput. It takes around 40 seconds for LWIP to transfer one frame worth of data at a packet size of 1500 bytes to the destination server. It would almost be impossible for the group to achieve video streaming over the network with simply the TCP connection.

Consequently, the group investigated into the UDP protocol which does not offer the same quality of transmission results but could transfer at a higher throughput due to lower overhead. However, the team discovered around 20% packet losses within each frame of data so additional headers need to be added for synchronization purposes. With UDP, the team achieved 1 FPS at 640x480x3 or 4 FPS at 320x240x3 resolutions which are enough to monitor the camera output in a timely manner over the network. In the final design, the group used TCP for handshakes such as “video start” and “video end” to the server and UDP for the frame data transfer. This ensures that both the client and server are synced with tasks and still achieving high enough throughput for video streaming across the network.

4.5 Digilent RGB to DVI IP

When switching to the Nexys Video board, the group discovered that the VGA port was no longer available on this board and the original TFT controller from Xilinx was no longer usable. To have a local video output for debugging and monitoring purposes, the group integrated into the HDMI output port and found the Digilent HDMI example project for the Nexys Video board. After fixing multiple build issues due to different versions of Vivado, the team managed to successfully deploy the HDMI example from Digilent onto the Nexys Video board. The group then removed all unnecessary components such as the HDMI input (as we are using the HD PMOD camera instead) and integrated all the other components into this framework. The RGBtoDVI core worked in parallel with the VDMA and the video timing controller to output HDMI signals at 640x480@60Hz from the memory. Having this capability allowed us to debug various issues with the camera module as well as comparing the output of the network frame transfer from the original frame.

5.0 Description of Design Tree

This section describes the GitHub file tree for this project. The project can be found on GitHub through the link below:

https://github.com/jiayuanchenece/G1_FPGA_Surveillance_System

```
G1_FPGA_Surveillance_System
 ┣ docs                                          # Project documentations
 ┣ host_server                                   # Remote server source
 ┣ src                                           # Vivado source files
 ┃ ┣ new_ip                                      # Custom motion detection IP
 ┃ ┣ proj                                        # Vivado project
 ┃ ┃ ┣ Nexys-Video-HDMI-master.cache             # Vivado cached files
 ┃ ┃ ┣ Nexys-Video-HDMI-master.hw                # Vivado hardware files
 ┃ ┃ ┣ Nexys-Video-HDMI-master.ip_user_files     # Vivado IPs
 ┃ ┃ ┣ Nexys-Video-HDMI-master.runs              # Vivado runs
 ┃ ┃ ┣ Nexys-Video-HDMI-master.sim               # Vivado simulation
 ┃ ┃ ┣ Nexys-Video-HDMI-master.srcs              # Vivado Verilog source files
 ┃ ┃ ┣ Nexys-Video-HDMI-master.sdk               # Vivado SDK
 ┃ ┃ ┃ ┣ RemoteSystemsTempFiles
 ┃ ┃ ┃ ┣ System                                  # Main SDK source
 ┃ ┃ ┃ ┃ ┣ src
 ┃ ┃ ┃ ┃ ┃ ┣ CameraControl                       # Camera control API
 ┃ ┃ ┃ ┃ ┃ ┣ DisplayControl                      # Display control API
 ┃ ┃ ┃ ┃ ┃ ┣ DynamicClockControl                 # Dynamic clock control API
 ┃ ┃ ┃ ┃ ┃ ┣ InterruptControl                    # Interrupt control API
 ┃ ┃ ┃ ┃ ┃ ┣ MotionDetection                     # Custom motion detection API
 ┃ ┃ ┃ ┃ ┃ ┣ Platform                            # Platform configuration
 ┃ ┃ ┃ ┃ ┃ ┣ TimerControl                        # Timer control API
 ┃ ┃ ┃ ┃ ┃ ┣ logo.h                              # Overlay logo
 ┃ ┃ ┃ ┃ ┃ ┣ lscript.ld                          # Linker file
 ┃ ┃ ┃ ┃ ┃ ┣ main.c                              # Main – entry point
 ┃ ┃ ┃ ┃ ┗ ┗ main.h                              # Main includes
 ┃ ┃ ┃ ┣ System_bsp                              # System board support file
 ┃ ┃ ┃ ┃ ┣ Makefile                              # SDK Makefile
 ┃ ┃ ┃ ┃ ┗ system.mss                            # System microprocessor software spec
 ┃ ┃ ┃ ┣ top_hw_platform_0
 ┃ ┃ ┃ ┗ top.hdf                                 # System hardware description file
 ┃ ┃ ┣ Nexys-Video-HDMI-master.tmp               # Vivado temporary files
 ┃ ┃ ┗ Nexys-Video-HDMI-master.xpr               # Vivado project file
 ┗ ┗ repo                                        # Digilent IP source files
```

6.0 Tips and Tricks

In this section, we would like to provide some time saving tips and tricks that hope to help the future students of ECE 532.

1. Vivado SDK would often report that the project is broken, or it could not build successful by mistake. This is especially common when you include the network LWIP libraries into your source code. To fix this issue, right click the application in Vivado SDK and select – clean project. Cleaning the project would clear the cache for existing object files and would often solve the issue.

2. When using Vivado to create the top-level wrapper for your block design, Vivado often does not update the wrapper when you modify the IO ports or memory sizes. To force Vivado in updating the wrapper file, right click the block design and select "Generate Output Product", this would often take a few seconds, but the wrapper file would be updated once complete. This is useful when you are manually writing the constraint file and are looking for the correct IO names.