Ma, Qian

Pak, Jonathan

Zahedi, Hamed

Zhu, Xingye

**Zynq-7000 Based Real-Time HDMI Display Processor**

**ECE1373H Final Report**

**8/24/2014**

**Professor Paul Chow**

Contents

[1. Introduction 4](#_Toc396658533)

[1.1 Overview 4](#_Toc396658534)

[1.2 Basic specifications 5](#_Toc396658535)

[2. Current Status 7](#_Toc396658536)

[3. Initial Design and Architecture 8](#_Toc396658537)

[3.1 Image Convolution Theory 8](#_Toc396658538)

[3.2 Initial System Design Architecture and Specifications 9](#_Toc396658539)

[3.3 Design Process and Brainstorm (System) 10](#_Toc396658540)

[3.4 Design Process and Brainstorm (Video Filter) 11](#_Toc396658541)

[4. Final Design and Architecture 13](#_Toc396658542)

[4.1 System Design and Architecture 13](#_Toc396658543)

[4.2 Overview of the data flow 14](#_Toc396658544)

[4.3 Implementation of image convolution algorithm 14](#_Toc396658545)

[4.4 Video Filter Design and Architecture 17](#_Toc396658546)

[4.4.1 Pipe stage and line buffers 19](#_Toc396658547)

[4.4.2 Shift FIFO 20](#_Toc396658548)

[4.4.3 Edge handling logic 20](#_Toc396658549)

[4.4.4 Multiplication and Recombination Calculations 20](#_Toc396658550)

[4.4.5 Other miscellaneous functions 20](#_Toc396658551)

[5. Register Configuration and Software Support 22](#_Toc396658552)

[5.1 Software Interface 23](#_Toc396658553)

[6. Methodology 25](#_Toc396658554)

[6.1 Design Methodology 25](#_Toc396658555)

[6.2 Design Environment 26](#_Toc396658556)

[6.3 Design Partitioning 27](#_Toc396658557)

[6.4 Verification Methodology 28](#_Toc396658558)

[7. Contributions 30](#_Toc396658559)

[7.1 Contributions: Jonathan Pak 30](#_Toc396658560)

[7.2 Contributions: Qian Ma 30](#_Toc396658561)

[7.3 Contributions: Xingye Zhu (Lucy) 30](#_Toc396658562)

[7.4 Contributions: Hamed Zahedi 30](#_Toc396658563)

[8. Design Characteristics 31](#_Toc396658564)

[8.1 Resource Utilization 31](#_Toc396658565)

[8.2 Project timeline 31](#_Toc396658566)

[9. Issues and Problems Encountered 32](#_Toc396658567)

[9.1 Design Problems 32](#_Toc396658568)

[9.2 Verification Problems 32](#_Toc396658569)

[9.3 Tool Versions 32](#_Toc396658570)

[9.4 Other 32](#_Toc396658571)

[10. Conclusion 33](#_Toc396658572)

[11. Appendix 34](#_Toc396658573)

[11.1 Line Buffer Detailed schematics 34](#_Toc396658574)

[11.2 Shift Fifo and Edge Handling Detailed Schematics 35](#_Toc396658575)

[12. Citation 36](#_Toc396658576)

List of Figures

[Figure 1‑1: High-level overview of application 4](#_Toc396658577)

[Figure 3‑1: Pictorial description of image convolution 8](#_Toc396658578)

[Figure 3‑2: Different kernels to create different effects 8](#_Toc396658579)

[Figure 3‑3: Initial Design architecture 9](#_Toc396658580)

[Figure 3‑4 : Initial filter design 11](#_Toc396658581)

[Figure 4‑1: Final system block diagram 13](#_Toc396658582)

[Figure 4‑2 – Line buffer implementation 16](#_Toc396658583)

[Figure 4‑3 - Shift FIFO operation 17](#_Toc396658584)

[Figure 4‑4: Final video filter design and block diagram 19](#_Toc396658585)

[Figure 4‑5 : Register and Clock Domain Crossing 21](#_Toc396658586)

[Figure 5‑1 : Software Configuration and Debugging 22](#_Toc396658587)

[Figure 5‑2 - C# based client-server interface over UART 23](#_Toc396658588)

[Figure 5‑3 - C#- based User Interface 24](#_Toc396658589)

[Figure 6‑1: Diagram showing the DUT and AXI4S TPG 28](#_Toc396658590)

[Figure 11‑1: This schematic shows more depth into the design of the line buffer. 34](#_Toc396658591)

[Figure 11‑2: Diagram depicts shift fifo and surround logic 35](#_Toc396658592)

# Introduction

## Overview

VLSI digital system design is vast and modern day System On-Chip (SoC) ASICS can perform a large variety of functions with embedded processors. In industry, these large scale designs are partitioned into smaller functional blocks (IP) and sometimes the overview of the entire system can be lost to engineers. However, the holistic view of the system is important to understand so it can help drive critical design decisions as well as understand system performance.

By undertaking this project, there is an opportunity to design an embedded system at a smaller scale within an FPGA which can help elaborate more intricate details in designing an entire system. For example, this project will start with what we hope to accomplish (i.e. display processing) which in turn will be translated to specifications regarding the necessary performance targets. The specifications will then be further developed into a higher level system diagram outlining all the necessary building blocks. Finally, a detailed architecture of the functionality can be elaborated after which implementation of logic can begin.



Figure 1‑1: High-level overview of application

The chosen project was to implement a configurable display processing engine which can apply a variety of image processing filters. The goals we wish to achieve for this project include:

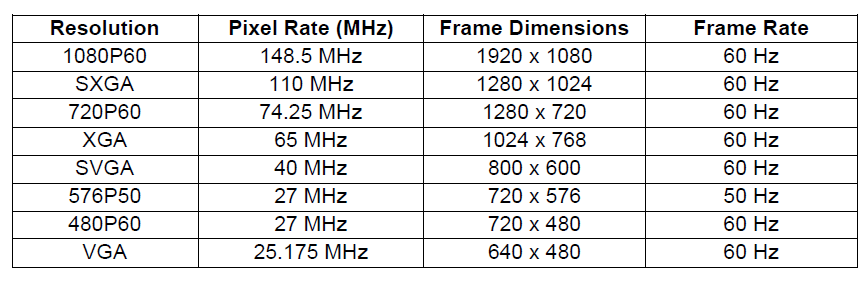
* Become familiar with FPGA embedded system design flow
  + Using a Xilinx development platform (Zedboard)
  + Demonstrate hardware capabilities through an FPGA embedded system
* Take a *live* video source and perform image processing to achieve a variety of effects such as sharpening, blur, edge detection, emboss, etc.
* Consider both hardware and software portions of an embedded system
* The display processing engine should be user configurable allowing the user to program different filters to achieve different image processing effects.
* The display engine needs to process pixels to allow for live video (streaming)

## Basic specifications

This project was implemented on a Zedboard (Zynq-7000) with an FMC Imageon HDMI transmitter/receiver add-on. The FMC Imageon card allows us to capture information from a source as well as display video onto a panel. Since the target is to implement streaming, this means we need to process pixels in real time and ideally at a rate of one pixel per cycle. One of the more popular standards today in display technology is 1080p (1920x1080 @ 60fps). To process this resolution, the pixel clock will need to be at least:

Along with some margin (20%), the pixel clock will run at approximately 150MHz to support 1080p resolutions. Resolutions smaller than 1080p can also be supported; the following table summarizes the resolutions that the design will support:

Table 1‑1: Supported resolutions and corresponding pixel rate



In terms of bandwidth, the system will only require one read and write port on the AXI interconnect to DDR memory. The DDR will act as the frame buffer for the system and is required because of the convolution algorithm that was implemented will introduce latency into the system. In normal operations, frames will be written to memory one by one but write and read operations will occur simultaneously as data fill and drain rates should be the same.

The pixel data from the FMC Imageon receiver module will be formatted in 16-bit YUV4:2:2 format. This is a compressed format where the Y (luma) component is 8 bits and the chrominance components (UV) are sub-sampled (also 8 bits). So the data is packed in the following order where each component is 8 bits:

The Zynq 7000 has 512MB of DDR3 and one 1080p frame only requires 4MB of memory so the amount of memory in the board is plenty. For software register access which does not need to run on a fast clock, the AXI-lite interface is used.

# Current Status

The system proposed was designed and implemented on Zedboard along with an FMC Imageon HDMI transmitter/receiver add-on. A demo was displayed on July 22nd, 2014 showing the operation of the system.

On the hardware side, all RTL is complete and intended functionality implemented. There are a few more enhancements which can be done:

1. Better parameterization of the RTL to allow easier switch to larger kernel sizes. Currently only 3x3 kernels are supported but transition to 5x5 (or larger) kernels will not be trivial.
2. Only image convolution was implemented but there are some other visual effects possible using other functions such as greater-than or less-than operators.
3. Addition of a hardware divider to remove software normalization for kernels. Normalization is required for kernels whose sum of elements is not unity because it will change the brightness value (Gamma) of the source image. Hardware division to accomplish gamma compensation is expensive, and performing this in software (normalizing kernel values prior to programming) should not incur any large overhead.
4. Consider improving precision of kernel values. Currently it is implemented as s4.5 (signed 10 bit number with 5 decimal bits). Needs more testing.
5. Cascading video filter IP to allow for multiple cascaded filters. This is not tested but in theory can be done.

On the software side, a driver was built to initialize all modules in the system, detect incoming video source and resolution, and drive data to the external display. A simple command line interface is used to configure the display processor and choose different kernels. The driver today is still running on the Xilinx SDK on the host machine and connected to the FGPA via JTAG. Some possible enhancements include:

1. Improving UI.
2. Remove dependency on the Xilinx SDK. I.e. porting the driver to run on Zynq 7000 ARM processor

# Initial Design and Architecture

## Image Convolution Theory

Image convolution refers to using a weighted sum of neighboring pixels to generate an output pixel. A square matrix known as a convolution kernel slides across every line of the frame to generate a new pixel. Figure 3-1 depicts the process whereby a 3x3 kernel convolves over a source pixel matrix to produce an output pixel

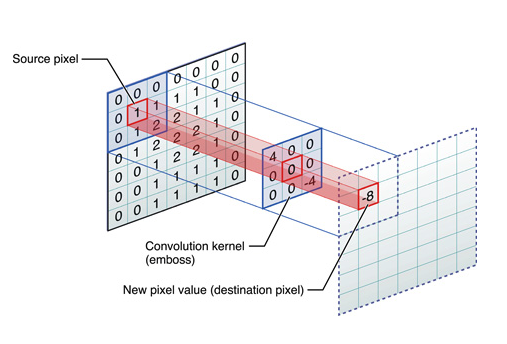


Figure 3‑1: Pictorial description of image convolution

By using different kernel values, different effects can be achieved. Figure 3-2 illustrates this. It is important to note that the only the luminance values of a color image will be operated on during the convolution. The chroma values are unaffected.

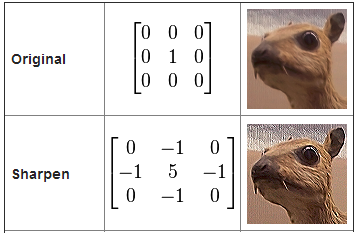


Figure 3‑2: Different kernels to create different effects

The display processing block will implement this image convolution to produce varying effects on the source video. Each frame will be convolved. The elements of the kernel will be programmable so the filter will be highly configurable.

## Initial System Design Architecture and Specifications

The following diagram shows the original system architecture:

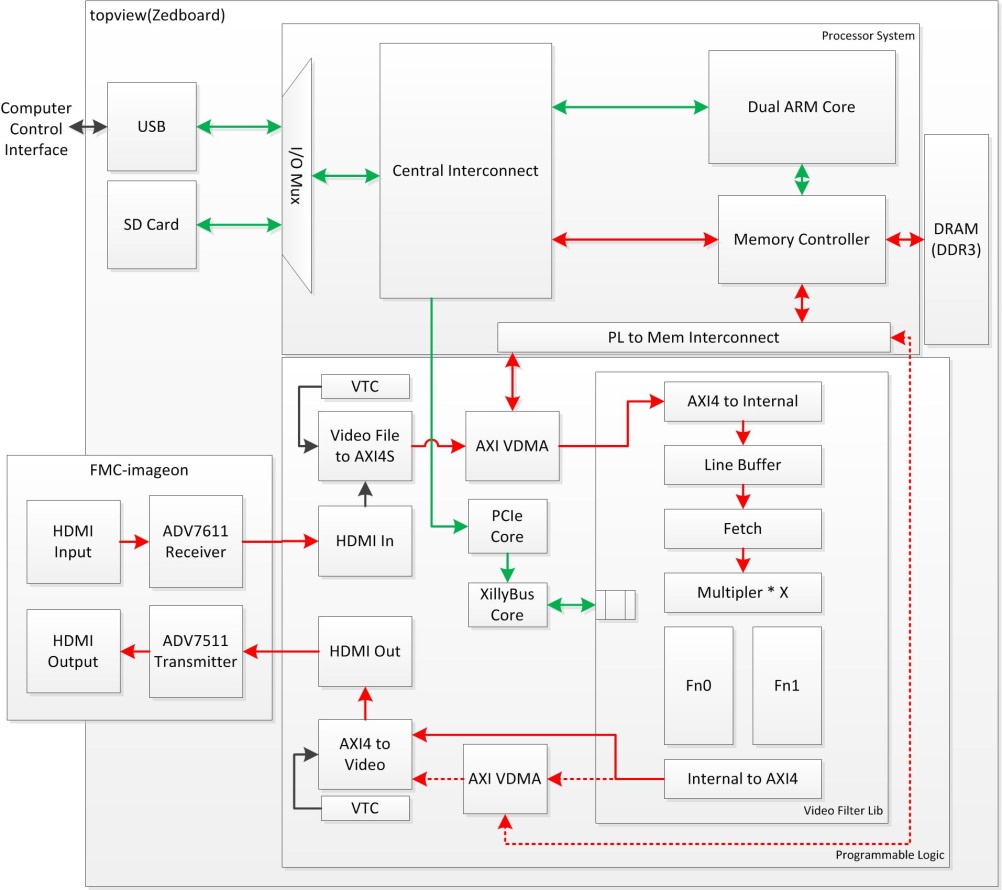


Figure 3‑3: Initial Design architecture

In general, there are two ‘sides’ to the design and the separation is at the Video DMA (VDMA) block which performs memory access. The modules before the VDMA block will be referred to as the input side and the modules after will be referred to as the output side. A short description of each block follows:

* **HDMI IN (receiver):** This is a third-party IP provided along with the FMC Imageon board and it provides an interface to the PHY. The output of this block is raw pixels with timing information (Vertical Syncs/ Horizontal Syncs).
* **Video File to AXI4S**: This module is a simple packing module which takes the incoming video data and extracts the active video to pack it into AXI4s (AXI for stream) format. The timing information is then passed to the VTC block.
* **Video Timing Controller (VTC):** There are two VTC’s within the design; one on the input side and another on the output side. The VTC on the input side is responsible for taking the timing signals from the video File to AXI4S module and detect the resolution/timing information. This information is then written to the registers within the block and also passed onto the VTC on the output side where the video data will be recombined with the timing information.
* **Video DMA (VDMA):** The VDMA block handles access to DDR3. Video data will be written and read from memory in order and one frame at a time. In the system diagram, there are two instances and the second instance on the output side is for debugging purposes. The intention is to read out the data in the frame buffer for debug.
* **Video Filter:** The video filter takes the incoming pixel data and performs image convolution to produce a modified image. By modifying the kernel parameters; it is possible to create different effects. This module is the main processing module within the system and more in depth description of this module will be provided in later sections of this document.
* **AXI4s to video out**: This block performs the inverse function as the video in to AXI4S block. It combines the regenerated timing signals from the VTC (output side) with the active pixel data from the AXI4s bus to re-create the video signals which are sent to panel.
* **HDMI out (transmitter):** Interfaces with the FMC Imageon card and transmits the pixels to panel
* **Xilly bus:** Since the video filter requires configurability, Xilly bus was originally included in the design to handle programming of software registers.

## Design Process and Brainstorm (System)

The design process began with researching which third party IP are available. Xilinx has a wide variety of third party IP available related to video processing. Xilinx also employs the AXI4stream interface which is a slightly modified version of the standard AXI interface. In addition to regular bus signals, the AXI4stream also includes start of frame (sof) and end of line (eol) signals which helps simplify logic downstream in regards to determining the position of the frame. With the exception of the video filter; the rest of the modules used in the design are Xilinx third party IP’s.

Using the AXI4s interface will simplify the processing modules’ design significantly since AXI4s only transmit active pixel data. The VTC blocks handle extracting and re-creating the timing signals on the input and output side respectively. Therefore, the video in to AXI4s and AXI4s to video out module allows us to take advantage of the AXI4s interface and this conversion will be placed before and after the video filter.

Implementing the image convolution algorithm will mean that we need to accumulate several lines of data before there are enough pixels to process. However, this will add latency to the system since a certain number of lines must be buffered before processing can begin. Therefore a frame buffer is required. Since several frames of data may be saved and the maximum resolution is 1920x1080 pixels; it is more efficient to use the DDR than block rams. The VDMA (video direct memory access blocks) are used as the memory controller and two were originally meant to be instantiated in the system; before and after the video filter. The VDMA after the DDR3 is meant as an optional debug feature where post-processed frames can be written back to memory and read from the processor.

## Design Process and Brainstorm (Video Filter)

The initial video filter design is depicted in the diagram below:

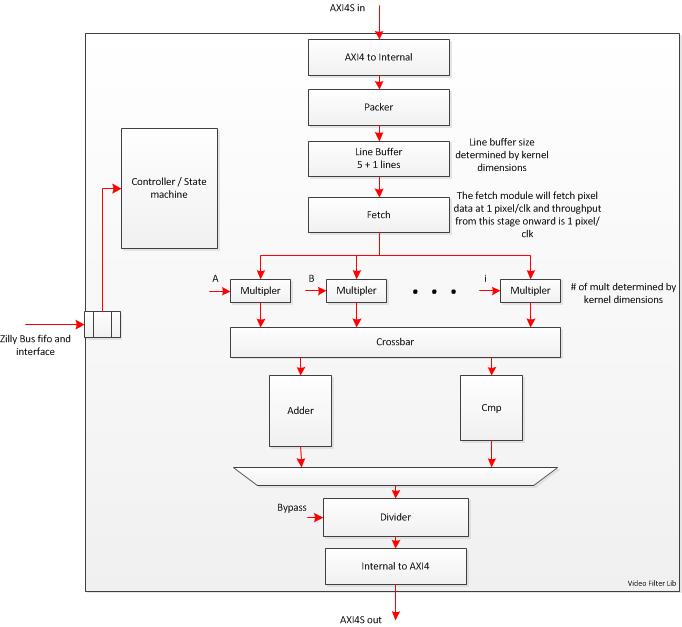


Figure 3‑4 : Initial filter design

The AXI4S to internal and packer modules were meant to extract only the Y component from the pixel data (which is encoded as YUV422) and pass it to the line buffers. The line buffers store 6 lines to support up to a 5x5 kernel. The fetch module reads from the line buffers and passes data to the multipliers at 1 pixel per clock. The multipliers will be implemented as fixed-point. The precision at this stage of the design was not set determined. To support a 5x5 kernel, a total of 25 multipliers are required.

Other than convolution, other functions can also be done and therefore the diagram shows both an adder block and a comparator block. A crossbar would be implemented to route data to the appropriate function. A divider was also added to normalize the pixel value after the multiplication. This feature is designed to provide gamma compensation - if the sum of the kernel elements does not equal 1; the brightness of the resultant pixel would have been increased and therefore would distort the output image.

# Final Design and Architecture

This section will describe the final architecture and design of the system and video filter portion of the design.

## System Design and Architecture



Figure 4‑1: Final system block diagram

Compared to the original system diagram, the main pipeline is very similar but some notable differences include:

1. **Removal of 2nd VDMA**: This debug feature originally planned was removed due to time constraint and it would not have provided enough value. Functional simulation would be sufficient to test the pixel values and reading the entire frame data back from DDR in hardware was unnecessary.
2. **Video Filter moved before VDMA**: The video filter in the original system diagram was placed after the VDMA. However in the revised implementation, the video filter is placed before the VDMA. It is likely that both methods would have been fine but moving the video filter before the VDMA should reduce the backpressure on the output of the video filter.
3. **Removal of Xilly bus**: Within the Xilinx SDK, it is possible to assign an address space to a custom PCORE. This is done almost automatically and a simple function call to write to a particular address will be sufficient to program these registers. This method is much simpler than instantiating and using the Xilly bus. Also, we were not planning to run an HLOS on the processing system.
4. **Asynchronous input and output sides**: The input and output side do not need to operate with synchronous clocks. So the input pixel clock and output pixel clock are asynchronous to each other but timing is closed on both at 148MHz.
5. **Moving to 3x3 Kernel** : We decided that the kernel size made little impact on overall architecture beyond increasing line-buffer requirements. A 3x3 kernel was sufficient for this project.

In regards to final specifications; it was not modified since the original. The maximum resolution remains at 1080p (1920x1080 @ 60 FPS) and the required pixel clock frequency is approximately 148.5MHz.

## Overview of the data flow

Pixel data flows from left to right in the diagram above. The input side and output side have asynchronous clocks. The two clock domains are separated by the VDMA module (write = input side, read = output side). Raw video data contains both pixel data and blanking periods (i.e. vblank and hblank). The active video data (i.e. pixels) are extracted from the video data stream using the video in to AXI4s module which outputs the pixel data using the AXI4s interface. The timing data is detected using the VTC (input side) and is saved into registers. The timing information is shared with the output side VTC to re-create it for the output video stream. The AXI4s active pixel data will get processed within the video filter and then the post-processed pixels are written into memory via the VDMA. The pixel data is drained from the frame buffer whenever it is available and requested by the panel. It will be re-combined with the regenerated timing signals from the output VTC and the AXI4s to video out modules.

## Implementation of image convolution algorithm

The image convolution algorithm requires surrounding pixels of the active pixel to calculate the new pixel value. The active pixel is defined as the current pixel being processed. However pixel data is provided in the order of the incoming scan line which means it needs to be stored in memory for processing. For a 3x3 kernel; 3 lines as well as 3 pixels of each line is required. For the remainder of this section; it is assumed a 3x3 kernel is used (and in actuality a 3x3 kernel is what is implemented) for ease of explanation. It is a simple extension for larger kernel sizes.

To implement this, a number of line buffers to store an entire line will be instantiated. The number of line buffers required is equal to the dimension of the kernel plus one because the extra line will be filled while the other ones are being processed to reduce the latency. For example, for a 3x3 kernel four line buffers are required. The control logic for the line buffers includes read/write pointers and counters to keep track of which pixel within the frame is the active pixel. The write pointer determines which line buffer incoming data is written to and the read pointer determines which line holds the active pixel. The pointers will wrap as the data is written/read and write collision flags will trigger when the write pointer is a certain distance away from the read pointer (depending on the size of the kernel, data cannot overwrite lines before and after the active line as the pixel data in these lines are still used in the calculation of the active pixel). Reading from the line buffer can begin as soon as two lines have been completely stored (assuming 3x3 matrix) and data is read from all line buffers except the one which is being written to. Edge handling logic will handle generate the virtual line before the first line of the frame hence it is not necessary to wait for three lines to be stored.

Several adjacent pixels in a line are also required in the calculation so the data from the line buffers will be pushed into a four-element shift FIFO (one per line buffer). The FIFO is designed to display the last three elements simultaneously but a read operation from the FIFO will only pop/shift one element out. This FIFO implementation combined with the line buffers will provide nine pixels per cycle for processing. The pixels will then proceed into the multipliers and adder to produce the modified pixel output.

Figure 4-2 below displays an example of possible pixel content within the line buffers. In this diagram, the line buffers fill from left to right and the pixel is identified by the two indices (the numbers represent x,y coordinates within the frame). In this example, the P0,0 pixel is the first pixel of the frame and it is stored in line buf2. The next two lines are stored in line buf3 and line buf0 (wrapped). Line buf1 is not used and data is being written. All line buffers will output data using the same address except for line buf1 since data is being written. Besides from pixel data, tracking signals such as SOF (start of frame) , SOL (start of line), EOL, and EOF (end of frame) are also written into the buffers. These signals follow the data flow to keep track of the position of the frame and will be consumed in the edge handling logic downstream.



Figure 4‑2 – Line buffer implementation

Figure 4-3 below, shows an example of pushing data into the shift FIFOs as well as how the frame’s upper right edge is handled. The RED highlighted cell is P0,0 and is the active pixel. The YELLOW highlighted cell represents the ROI (region of interest) and these are the pixel values which will be processed (note a read pointer is used to keep track of where the active pixel is). The first pixel of a line is a special case since the logic always requires the active pixel to be the middle element within the matrix; hence that the first column within the shift FIFOs must be pre-filled with dummy data so the first pixel gets written to the second column. Edge handling is done by cloning the closest pixel to the edge. Note that four FIFOs were used even though only three FIFOs would have been sufficient. This is to reduce the multiplexing logic between the line buffers (block rams) and the shift FIFOs as access time from rams can often be problematic for timing closure.



Figure 4‑3 - Shift FIFO operation

## Video Filter Design and Architecture

The video filter design has been simplified and streamlined compared to the initial version. A few features which were originally planned were dropped and these include:

* **Multifunction algorithm block**: Originally, functions other than multiplication were planned for implementation. Due to time constraints this has been dropped.
* **Divider**: The divider’s purpose is to normalize the kernel to not distort the pixel value but adjusting the original brightness. However, a divider is expensive to implement in hardware and not trivial. The benefits of a hardware divider in this use case are also not warranted as the kernels can be pre-normalized before programming anyways. For this reason, the divider is dropped and it is expected that the kernels be normalized by the user before programming the video filter.
* **Misc changes:** The fetch/packer modules in the initial diagram are not required. The AXI4s interface provides only active pixels so there is no need to format/pack the data another way. The Y component is all that is required for the filtering so this component is extracted from the raw pixel data (8 MSB of the 16 bit pixel data) and processed. The other Cr/Cb components are only passed through the system with no processing and recombined at the end of the pipe.

The new/revised modules within the final implementation are:

* **Pipe state (5 stages)**: Five pipe stages were added to the beginning of the video filter and it provides 5 cycles of delay. The purpose of this pipe stage is to provide enough time for the video filter to re-initialize pointers/counters between frames. The AXI4s interface only provides SOF and EOL signals but not EOF. EOF is generated internally and is used to tell the design when the current frame is finished and to reset the logic. Since the re-initialization takes a few cycles, the pipe stage at the beginning of the pipe provides the time to do so.
* **Shift FIFOs:** For more details on the shift FIFO implementation, refer to the previous section.
* **AXI4lite address decoder and software registers:** This module holds the software registers to control the video filter including the kernel elements. Other register controls include the debug bus selection and bypass mux control.
* **Filter ctrl/FSM:** This module contains FSM and control logic for the video filter. It generates reset signals to the pipes once a frame has been processed. It also controls when soft controlled signals can be applied to the video filter. For example, the controller only allows the bypass signal to reach the logic between frames and not in the middle of a frame (necessary to avoid corrupting the current frame).
* **Debug**: The debug module is meant to generate signals for debug. It latches the first occurrence of some key signals for debugging the system after it has been implemented. It also multiplexes several debug buses from different modules onto the software registers.

Figure 4-4 below depicts the final revision of the filter design. The data flow through the video filter is straightforward and flows from top to bottom. Note that only the luma (y) component of the pixel data is used for processing while the Cr/CB components are simply passed through. A short description of block operations is as follows



Figure 4‑4: Final video filter design and block diagram

## Pipe stage and line buffers

Active pixel data as well as control signals (sof, eof, sol, eol) from the AXI4s interface will first go into a pipe stage (latency of 5). The line buffers will accumulate two full lines of pixels before data is drained from the buffers. There are 4 line buffers and the number of line buffers required is equal to the kernel size plus one (i.e. 3+1). Three lines must be maintained since all three lines are used in the calculation of a single pixel. The fourth line will be filled while the other three are used for processing to eliminate latency. A minimum of two lines are required because the video filter processes data using a kernel with maximum size of 3x3. The minimum number of lines required is equal to the dimension of the kernel minus one (i.e. 2) since the first two lines are enough to calculate the first pixel (this is true since edge handling will ‘generate’ the n-1 line). The consequence of accumulating two lines before any calculation is that a two-line latency is imposed on the system. This does not pose a problem since two lines can be absorbed in the blanking periods of the video stream. All four line buffers will be read from at the same time and the three valid lines from each line buffer are selected via a read pointer. The valid lines are then written into shift FIFOs.

## Shift FIFO

The shift FIFO will accumulate pixel data where the line buffers accumulate entire lines. The shift FIFOs are four entries deep (same reasoning as number of line buffers i.e. kernel size plus one). The output of the FIFOs will read the last 3 entries and a read operation will only pop the last entry. Extra logic was implemented to always pre-fill the first entry of the FIFO (if no valid data is available, i.e. before the first pixel of the frame, dummy data will be written to pre-fill the FIFO). This is to aid in tracking the position of the active pixel and to keep the active pixel in the centre of the square matrix. Edge handling logic will mask any invalid/dummy data. Note that besides pixel data, tracking signals such as sof, eof, sol, and eol are also pushed into the FIFOs. Total of 9 pixels output from the shift FIFOs and all are used in the convolution with the 3x3 kernel.

## Edge handling logic

sof, eof, sol, and eol are tracking signals which follow the first pixel of each frame, last pixel of each frame, first pixel of each line, and last pixel of each line respectively. Some of these are provided by the AXI4s protocol (sof, eol) while eof and sol are internally generated. These signals are propagated through the line buffers and shift FIFOs so it can be used by the edge handling logic. By reading these signals, the edge handling logic will know which pixel to extend. For example, if a pixel is accompanied with sof then edge handling will extend the first column and first row.

## Multiplication and Recombination Calculations

The nine pixel data from the shift FIFOs/edge handling logic are each multiplied by one of the nine corresponding kernel elements which have been programmed from software. This is done in parallel and precision is s4.5 (1 signed bit, 4 integers, 5 fractional bits for a total of 10 bits). The resultant nine products are then summed to generate the final pixel value. The pixel is then clamped to an integer and negative values are set to zero. The luma value is then recombined with the original Cr/Cb value to regenerate the entire pixel. The entire pipeline uses a ready/valid handshake protocol which is then passed onto the AXI4s interface downstream.

## Other miscellaneous functions

Besides the main pipeline, there are other modules with the video filter such as the controller, debug module, and software register module. The controller contains FSM’s which control the state of the pipeline as well as when to restart the pipe. It also polices when static configuration registers such as bypass enable and the kernel values can be updated (i.e. always after eof and never during the frame.

The software register module contains a decoder for the address and operates primarily on the AXI4lite domain. One ready signal is synchronized onto the Pclk domain and is used to gate the static configuration registers to ensure all values are stable before updating it into the main processing clock.

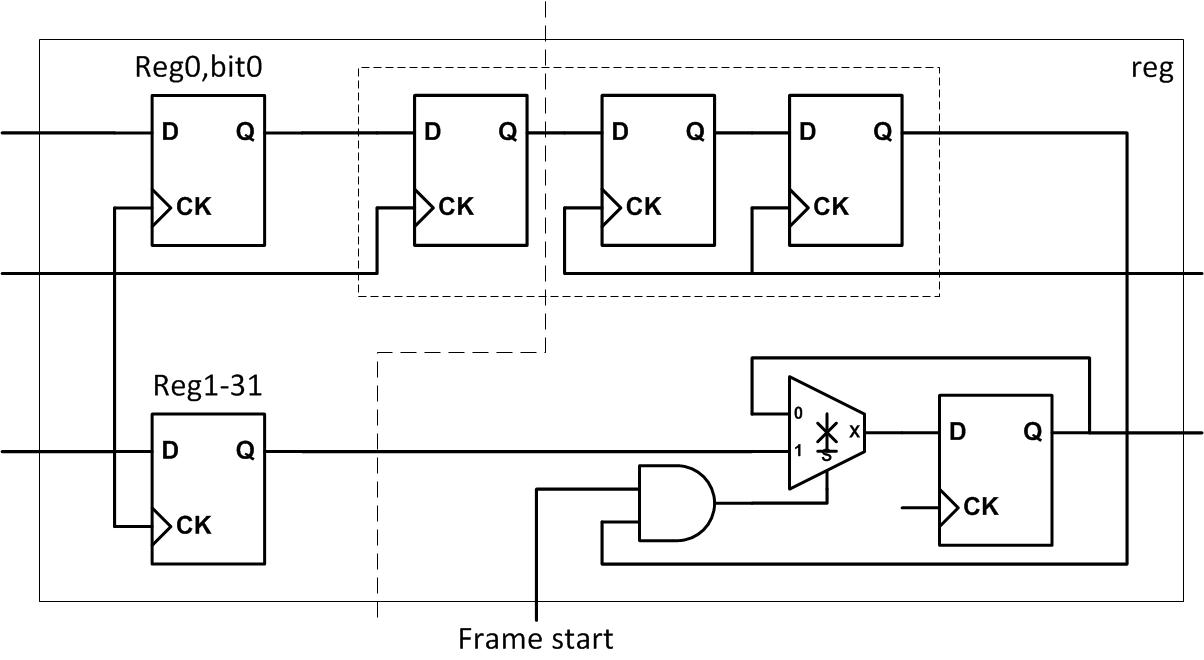


Figure 4‑5 : Register and Clock Domain Crossing

The debug module is a simple multiplexing block which allows the user to latch helpful debug signals and save it to software registers to be read out by software. This feature is event driven and therefore there are no strict timing requirements. It is invaluable in debugging the hardware in hang situations.

# Register Configuration and Software Support

The video filter’s configuration logic leverages the Xilinx AXI4-Lite PCORE design flow, which provides a direct programming interface from ARM A7 core to the user defined registers. To simplify the design procedure, a BAR address is allocated in the system when the registers are generated. By reading from/writing to the address (BASE + REG\_INDEX\_OFFSET), users can lively modify video filter configuration through Xilinx SDK, for example switching between function and bypass mode or updating the convolution kernel. The diagram below shows the overview of interaction between software and hardware.

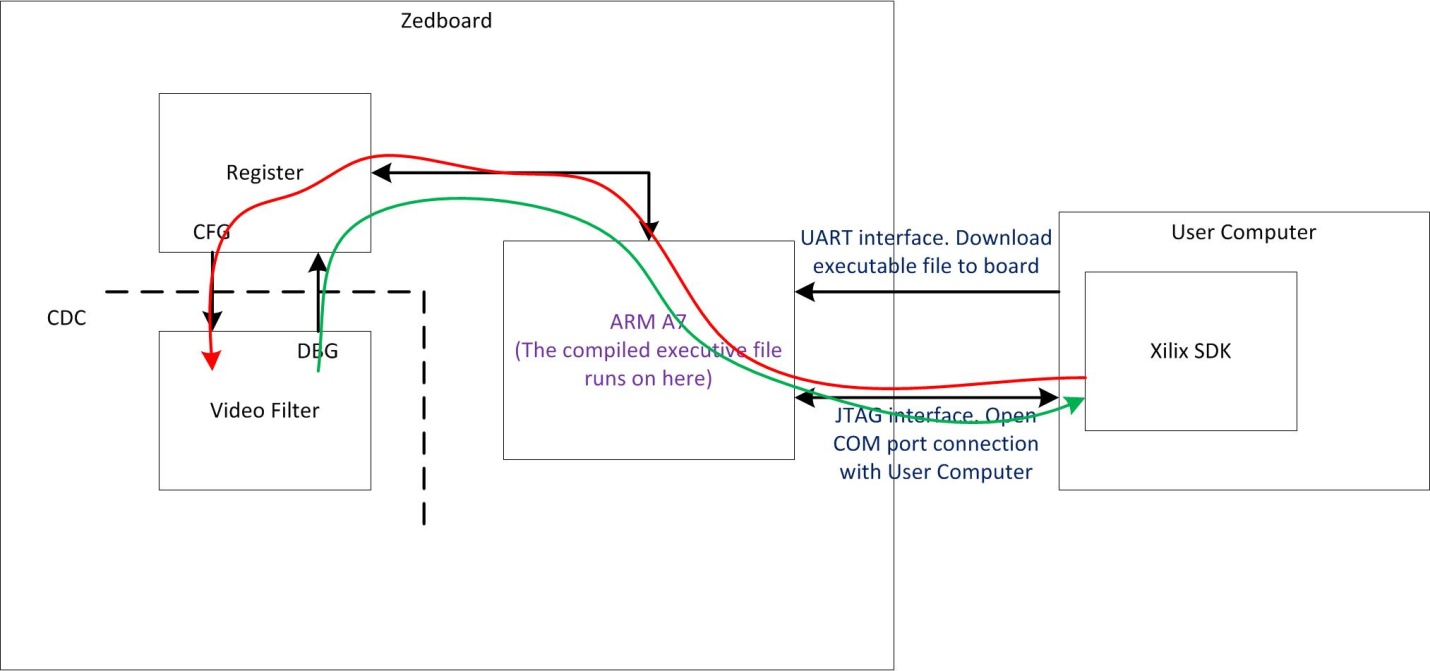


Figure 5‑1 : Software Configuration and Debugging

The configuration steps are listed below.

1. At the beginning, ARM core initializes the system through CFG interface using default setting (bypass) and in standby mode to listen to user input.
2. Users modify the configuration through JTAG interface. ARM A7 re-programs the registers.
3. If switching to debug mode, selective video filters’ registers are read back through DBG interface.

## Software Interface

Alongside the command-line interface developed through the Xilinx SDK, a C# based GUI was also developed to communicate through the UART interface to the Zynq 7000 FPGA from a Windows based Host PC. The software was written using Microsoft Visual Studio .NET 2013 and utilized the rich set of Serial Port library functions exposed by the C# platform.

Figure 5-2 depicts the high-level client-server model implemented partly on the C# GUI ( Server ) and the Xilinx SDK ( Client ). The Host PC initiates a basic handshaking mechanism to determine when the Zynq is connected and ready to receive commands from the command engine.

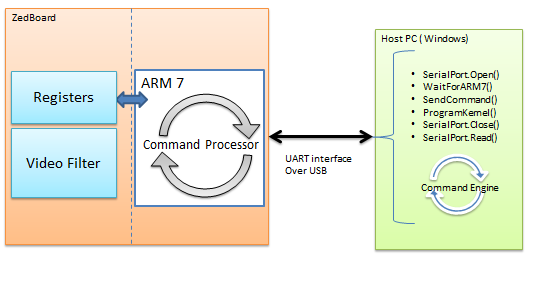


Figure 5‑2 - C# based client-server interface over UART

Once the link is established, the GUI allows the user to select a kernel from a drop-down list, or create a new kernel by editing the matrix. Once the user selects the “Program” button, the software calculates the normalization value ( shift value ) and packages the 9 coefficient values into register write command to be sent over the UART serial interface. This command is processed by the ARM 7’s Command processor and written to the registers to implement the required filter. A screenshot of this GUI is shown in Figure 5-3 below. The low-level details of the UART protocol are abstracted by the C# serial Protocol library on the Host Side and also by the Xilinx SDK’s console interface to the UART module on the Target FPGA.

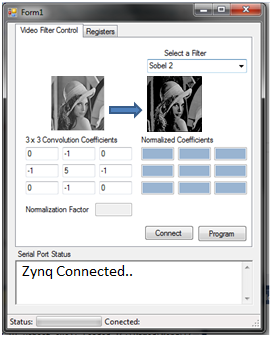


Figure 5‑3 - C#- based User Interface

During initial design, it was planned to have PetaLinux running on the Zynq 7000’s ARM processor with the entire GUI implemented on the target side, however this proved to be a difficult challenge given the timing constrains and we resorted to developing the interface on the Host-PC side with the client side residing on the ARM-7.

# Methodology

## Design Methodology

To implement a working prototype of the video filter, the first step is to determine the feasibility of the project given the specifications and hardware. There are different aspects of the design including:

1. Understanding the algorithm (i.e. image convolution)
2. Understanding the limitations of the hardware
3. Understanding current technologies and methodologies regarding display and video processing include different pixel formats.

For the first point above, a Matlab model was created to experiment with the algorithm. The model takes a source image and processes each pixel with different kernel types. This exercise allows us to determine and visually verify the effect of each kernel. The different types of edge handling algorithms were experimented on in the Matlab model as well to help determine the effect of each and which was most suitable. A review of the Zyng-7000 FPGA was also done to ensure there are enough resources to implement the video filter at the desired specifications.

A system level data flow diagram was brainstormed. At this stage, research was done on third party IP’s which can be used to help design the system. Xilinx provides a very large library of video processing components. The AXI4s interface was chosen to carry the data between modules at this stage. In addition, a pass through system (i.e. without the video filtering) was designed first to get the basic dataflow implemented and verified.

The next stage includes architecting the video filter itself and translating the image convolution algorithm to hardware. The algorithm is then partitioned into standalone functions and mapped to Verilog modules such as the line buffer, shift FIFOs, and multiplier.

Finally, a detailed draft of the system and video filter was drawn showing all interfaces between modules and estimated resources of each module were calculated.

## Design Environment

A variety of tools and platforms were used in the completion of this project.

#### Hardware:

* Zedboard with Zynq-7000
* FMC Imageon HDMI Transmitter/Receiver
* Laptop (video source) with HDMI output
* Any display panel for output (TV/computer monitor)

#### Software:

* **Matlab**
  + Matlab was used to generate a functional model to validate image convolution algorithm. The images produced provided samples for each of the different kernels.
* **Xilinx Planahead (Version 14.7)**
  + This is an integrated environment encompassing the EDK (embedded processor design and configuration) and the SDK (driver and software development). Hardware is integrated within Planahead and synthesis/place and route was done here as well. The software also allows all files to be organized effectively (i.e. hardware, third party IP’s, driver code, etc. are all managed within Planahead).
* **Xilinx Vivado HLS**
  + Experimentation with HLS tools for the Matrix Multiplier block
* **Text editor (gvim, etc)**
  + Any text editor was sufficient for the actual Verilog coding. Code is then imported into Planahead for integration
* **MS. Visual Studio 2013**
  + Used to develop a prototype a C# based GUI for interacting with the Xilinx SDK on the host system through UART interface

#### Verification tools:

* **Synopsys VCS MX**
  + Tool used for RTL simulation. This is an event driven simulator.
* **Isim** 
  + Xilinx simulator integrated within Planahead. While this tool is not used explicitly, the simulation models and testbenches are compatible with ISIM.

#### Source control:

* **Github**
  + Github was used as the primary method of source control.

## Design Partitioning

Other than third party IP’s, the only design element which required detailed design partitioning is the video filter. Within the entire system, the video filter was placed before the VDMA (i.e. on the input side) to reduce backpressure on the video filter and because the video filter introduces at least 2 lines of latency into the system. Placing it before the VDMA/frame buffer will help hide the latency. It was also decided to use the AXI4s interface which is a small modification to the standard AXI4 interface. In addition to the regular ready/valid/data signals, the AXI4s interface also provides tlast (eol) and tuser (sof) which helps in tracking the current position of the frame.

Within the video filter itself, the design elements are partitioned mainly by functions. The following are the modules within the video filter:

* Filter\_pipe\_stage
* Filter\_ctrl
* Filter\_lbuf
* Filter\_FIFO\_top
* Filter\_multiplier\_model
* Filter\_debug

This partition was chosen primarily to group similar logic functions together and to create a modular design. Typically, data flows from one module to the next and uses a standard valid/ready handshaking interface.

## Verification Methodology



Figure 6‑1: Diagram showing the DUT and AXI4S TPG

Pre-implementation hardware verification was done using RTL simulation. However, since many of the blocks in the system were third party IP; it was hard difficult to simulate the entire system. Rather, an AXI4s test pattern generator (tpg) was created to simulate different traffic conditions. AXI4 stream traffic is simple to model as it only consists of active pixel data and two handshaking signals, and sof/eol. Another advantage of this approach is that we can reduce the size of the design under test (DUT) and isolate it from the system as it is the primary design of interest for verification. In addition, the reminder of the system less the video filter can be implemented and tested easily as it will function as a simple pass through. The AXI4s tpg is a simple Verilog model which can be parameterized to mimic different traffic scenarios including different number of pixels and de-assertion of valid during and between lines. The AXI4s tpg generates incrementing data which restarts after every line. Hence the traffic into and out of the video filter is predictable and the final outputs can be calculated and matched with the simulation output.

Post-implementation hardware verification refers to testing the hardware with real video sources and a display panel. In this case, the FPGA is a blackbox and the use of the debug module was critical. The debug module within the design interfaces with the AXI4lite register interface and allows software to access values on the debug bus. In reality, the system is usually in a hang state so the timing of these signals is not important (hence no real need for CDC). The debug module also implements some counters (such as pixel, line, and frame counters) which can provide an indication of when the failure or hang occurs. The control module’s FSM states are also provided onto the debug bus. Since there are more debug signals than there are registers; a simple muxing scheme was implemented to allow different sets of debug signals to be read out (this is also programmed via software). In summary, when a hang or failure occurs (usually indicated by a still or blank image on the display), the debug information helps provide the following:

1. An indication of where in the frame the failure/hang occurs (i.e. first pixel, last pixel, last line, or between frames)
2. Which module is causing the hang. Each module within the video filter IP has a debug bus which provides status of its operation.

Once this information has been gathered, the design is reviewed for detection of the bug. When a possible bug has been identified, the test is updated to simulate the failure conditions (usually done by changing the traffic from the AXI4s interface). A fix to the RTL is implemented and re-implemented on the FPGA so the cycle can begin again.

# Contributions

This section describes the contribution of each member of our team. Much of the project was a group effort and inputs from everyone were valuable even if a particular element was designed or implemented by a single person.

## Contributions: Jonathan Pak

Jonathan was involved with the development of the algorithm and system architect. He wrote the Matlab model and used it to experiment on image convolution. He also wrote the RTL for the line buffer, controller, AXI4s test pattern generator module, debug module, and multiplier block within the video filter. Finally, he was also involved with the hardware integration of the system and the video filter itself.

In regards to verification, Jonathan worked on both pre-implementation and post-implementation bring up of the design.

## Contributions: Qian Ma

Qian was involved with the system architect and software architect. He wrote the RTL for the register and clock domain crossing logic. He also wrote the system control and configuration software. Finally, he was also involved with the hardware integration of the system and software bring up.

In regards to verification, Qian worked on both pre-implementation and post-implementation bring up of the design.

## Contributions: Xingye Zhu (Lucy)

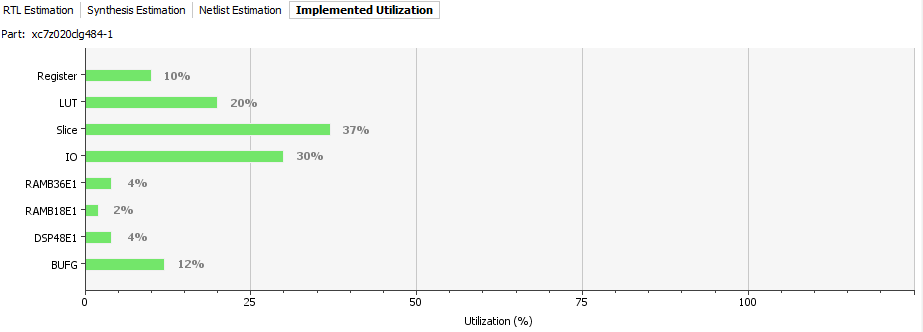
Lucy played a big role in assisting with hardware architecture and project management. She kept the team on track of its progress and made sure all the components to the project were ready. She also evaluated different options on implementation of the software.

## Contributions: Hamed Zahedi

Hamed was involved with project proposals, brainstorming and feasibility discussions. He evaluated the HLS tools on Vivado for the Matrix Multiplier. He was also involved in host side software development for the C# based GUI

# Design Characteristics

## Resource Utilization



There weren’t anything in the resource utilization charts that were not as expected.

## Project timeline

In general, the project definition, brainstorm, and learning process proceeded according to schedule. The HDMI pass through was implemented early on to validate the basic datapath. RTL coding and hardware implementation of the video filter also went according to schedule with majority of the RTL implemented by late April 2014 (3-4 weeks).

The post-implementation testing of the design proved to be much more timing consuming than originally expected. RTL was revisited several times as some design assumptions were proven wrong (we assumed there wouldn’t be backpressure and the panel can consume pixel as fast as it is provided) and bugs surfaced which were not caught in the pre-implementation testing. All of the bugs and issues found were related to specific traffic conditions and corner cases which require an update to the AXI4s TPG to generate the problematic traffic scenarios. This process was very time consuming because visibility into the system was limited (only debug information from debug bus was available) and synthesis/PNR on the FPGA takes approximately 40-50 minutes per spin. Only one FMC HDMI-IN/OUT board was available so testing was done by one person only and cannot be parallelized.

# Issues and Problems Encountered

## Design Problems

Given limited resources on board, one of the biggest design challenges is to support 1920x1080 live streaming, which requires at least 4x1920 16-bit RAM blocks to buffer the incoming data. Our strategy is to use high density memory instead of flops in design. Since memory doesn’t present the stored value one cycles after read enable asserts, we introduced shift FIFO to pre-fetch the data from the RAM based line buffers.

## Verification Problems

Expanding the simulation environment to cover the entire system would have helped catch some issues earlier.

## Tool Versions

Our project was started with Xilinx Planahead version 14.7 since many of the AVNet HDMI module IP integration tutorials and TCL automation scripts were written for this tool version. However, we also experimented with Xilinx Vivado 2013.2/2013.4 in order to use the Vivado HLS tools for algorithmic portions of the design. As such, we attempted to recreate the base AVNet IP integration steps in Vivado in efforts of porting the entire system to the newer platform – which boasts a significantly improved tool flow from design to validation to SW development and debug. However, it became apparent that there are no simple mechanisms for this migration and we were forced to abandon efforts and revert back to Planahead.

## Other

The post-implementation verification time would have been reduced if there were more FMC HDMI In/Out cards available. With only one card, only one person was debugging the hardware at any time which slowed down the process.

# Conclusion

Rapid FPGA development tools, methodologies, IP libraries and verification flows are affording system designers and integrators with an unmatched ability to prototype large system designs with efficiency, while allowing software and firmware developers with a flexible development platform. With the aid of these tools, and with the relatively low-cost hardware of the Zynq 7000 SOC-FPGA platform, we were able to successfully implement, and demonstrate the application of real-time image processing algorithms onto a live 1080P HDMI stream.

While the project team members have working backgrounds in ASIC design, verification and power analysis, and utilize a vast array of ASIC design, verification and simulation tools to develop commercial IP cores, this FPGA project provided exposure to possible areas where rapid FPGA system prototyping would be of tremendous value especially in situations where system-level simulation times would run into days or weeks for a single HD frame on traditional platforms. The tradeoff here was that we sacrificed signal level visibility in a simulation environment for the speed of a FPGA prototyping platform.

# Appendix

## Line Buffer Detailed schematics



Figure 11‑1: This schematic shows more depth into the design of the line buffer.

The line buffer module contains 4 block ram instances each 1920x16 (maximum pixel per line is 1920 for 1080p resolution and pixel data is 16 bits per component). It also contains the necessary control and pointer generation logic. The write pointer determines which of the block rams is free for writing. Data is read from all rams except the one which is being written to (i.e. this means three lines of data gets read). The read pointer keeps track of the active line where the active pixel is (always the middle line).

## Shift Fifo and Edge Handling Detailed Schematics



Figure 11‑2: Diagram depicts shift fifo and surround logic

The four instances shift fifo helps accumulate pixels (where as the line buffers accumulate lines). Data from the line buffers are written directly into the shift fifos. When filled; the fifos will drain and output three pixels at the same time. Also, note that the active pixel is always considered the center pixel within the matrix. For the corner case of the first pixel; the first entry in each of the fifos are pre-filled with dummy data so that when the first true pixel data is written into the fifos it will populate the second element.

The edge handling logic handles extending the pixels to handle edge and corners of the frame. It uses the sol, sof, eol, and eof signals generated further upstream to determine when edge detection is required.

# Citation

Kirshna, Vasmi. Kumbhare, Pankaj. (2014, Feb. 26). AXI VDMA Reference Design. [Online]. Available: http://www.xilinx.com/support/documentation/application\_notes/xapp742-axi-vdma-reference-design.pdf

Xilinx. (2013, Jan 03). FMC-Imageon Building a Video Design from Scratch. [Online]. Available: http://www.zedboard.org/sites/default/files/design/FMC\_IMAGEON\_Building\_Video\_Design\_Tutorial\_14\_4\_20130110.zip

Xilinx. (2014, April 2). AXI4-Stream to Video Out V3.0. [Online]. Available: http://www.xilinx.com/support/documentation/ip\_documentation/v\_axi4s\_vid\_out/v3\_0/pg044\_v\_axis\_vid\_out.pdf

Xilinx. (2014, April 2). Video In to AXI4-Stream V3.0. [Online]. Available: http://www.xilinx.com/support/documentation/ip\_documentation/v\_vid\_in\_axi4s/v3\_0/pg043\_v\_vid\_in\_axi4s.pdf

Xilinx. (2012, Oct 16). Zedboard: Zynq-7000 AP SoC Concepts, Tools, and Techniques: A Hands-on Guide to Effective Embedded System Design. [Online]. Available: http://www.zedboard.org/sites/default/files/design/ZedBoard\_CTT\_v14.5\_130808\_0.zip