NTNU

Norwegian University of
Science and Technology

Design and analysis of a video scaler suited for FPGA implementation

Tom Erik Tysse

Master of Science in Electronics
Submission date: June 2017
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Milica Orlandic, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Problem Description

Design and analysis of a video scaler suited for FPGA implementation

Video scaling is the process by which an input color image is converted to an output color image of a different size. The Video Scaler module is supposed to provide scaling solution for up/down scaling.

Furthermore, the implemented Scaler module is supposed to be incorporated into AXI-Stream Vivado Video system (containing Test Pattern Generator, VDMA modules, DDR memory, HDMI/VGA sink module, etc) and to be tested for various resolutions and frame rates.

Assignment given: 15. January, 2017
Supervisor: Professor Kjetil Svarstad, IES
Co-Supervisor: Postdoctoral Fellow Milica Orlandic, IES

Sammendrag

En video-skalerer, er en module som tar inn et bilde, forstørrer eller forminsker det, og det sender tilbake. Video-skaleringsmodule i denne oppgaven ble laget for å være en del av et AXI-Stream video-system som skal være tilpasset for å kjøre på en FPGA.

Systemet kan deles i tre deler: en video-skaleringsmodul, en videomodul og en kommunikasjonsmodul. Hovedoppgaven til systemet er å hente data fra minne ved hjelp av kommunikasjonsmodulen, for å så sende det til video-skaleren, som da ville skalere oppløsningen opp eller ned, for så i sende det nye bildet enten tilbake til minne eller til video-modulen, som tilpasser bildet til HDMI/VGA-format slik at det kan vises på en skjerm.

Video-skalereren bruker en algoritme for å firedoble antall piksler i et bilde. For å gjøre dette trenger skalereren informasjon om bildets bredde og høyde, ellers vil bildet bli forvrengt eller forskjøvet. Skalereren kan lett tilpasses bilder med andre mål.

For å nedskalere bilder, brukes en form for sammenslåing av 4 og 4 piksler om gangen. Både oppskalereren og nedskalereren ble testet på de samme bildene, og de har samme grensesnitt ettersom de skal kunne tilpasse seg et større system uten at det påvirker resten av systemet.

Oppskalereren ble kun brukt til simulering, mens nedskalereren ble brukt både til simulering og implementering. Systemet som nedskaleringsmodulen ble satt inn i kunne kommunisere med minne da det ble kjørt på en FPGA.

Summary

A video scaler, is a module that receives a picture, enlarge or shrink it, and sends it back. The video scaler in this thesis was made to be part of an AXI-Stream Video System that is suited to be implemented on a FPGA.

The system is divided into three parts: the video scaler module, the video module and the communication module, where the final goal was to use the communication module to send data from memory, to the video scaler, scale the video up or down, send it either back to memory or to the video module where it could be displayed on a screen by a HDMI/VGA-port.

In the video scaler, an algorithm was used to scale the images up or down by a factor of 4. The scaler needs to know the pixel-width and pixel-height of the image prior to the scaling in order to prevent the post-scaled image from being distorted or askewed. Width and height can easily be configured to fit new images.

For the downscaling, a form of merging was used to reduce the number of pixels. It was tested on the same images as the upscaler, and both of the scalers use the same interfaces since they have to fit into the same larger video system.

The downscaler was simulated and implemented into a system where it communicated with memory, which was running on the FPGA, while the upscaler was only simulated.

Acknowledgement

I would like to thank my supervisor, Professor Kjetil Svarstad as well as my co-supervisor, Post-Doctoral Fellow Milica Orlandic for all their guidance and help during this project. I would also like to thank my fellow students for engaging in many productive conversations.

This Master's thesis is the final product of a master of science in electronics at NTNU and a bachelor of engineering in automation technology from Bergen University College and I would like to thank everybody that has helped me on the journey.

Tom Erik Tysse

Contents

List of Figures

List of Tables

Abbreviation

FPGA Field Programmable Gate Array

DMA Direct Memory Access

VDMA Video Direct Memory Access

AXI Advanced eXtensible Interface

AMBA Advanced Microcontroller Bus Architecture

IP Intellectual Property

HDMI High-Definition Multimedia Interface

PS Processor System

PL Programmable Logic

FIFO First In First Out

MM2S Memory-Mapped to Stream

S2MM Stream to Memory-Mapped

DDR Double Data Rate

MIG Memory Interface Generator

BRAM Block Random-Access Memory

PNG Portable Network Graphics

RGB Red Green Blue

YCbCr Luma Chroma Blue Chroma Red

EPX Eric's Pixel Expansion

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

LUT Look Up Table

VTC Video Timing Controller

FF Flip-Flop

BUFG Global Buffer

XMD Xilinx Microprocessor Debugger

WNS Worst Negative Slack

TNS Total Negative Slack

Chapter 1

Introduction

1.1 Motivation

A video scaler is used to convert signals from one display resolution to another, either to a higher or a lower resolution. Take television for example, where video is broadcast to consumers who have televisions with different dimensions. To make the resolution of the video fit the display of each television, the images must be scaled or/and cropped locally in each TV.

Video scalers are often used in combination with other video processing devices or algorithms to improve the apparent definition of the video signals. Some the improvement techniques are encoding, digital filtering and scan interpolation, Bovik [5]. The final AXI4-Stream video system is intended to be a part of a Post-Doctoral project where some of these techniques will be present.

Since video is a sequence of images, which should be scaled rapidly, the algorithm that is implemented needs to be efficient as it should display a sufficient amount of frames per second and a satisfactory result on each frame rather than a few frames per second with 4K quality (normally referred to as lag) or 60 frames per second with a distorted image.

The FPGA, which the system has been implemented onto, is a integrated circuit developed to be reconfigurable and the board that has been used is ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. The reason for using a FPGA is that it is significantly faster than other processors for some specific tasks, because of its use of parallelism as well as optimizing the number of gates it uses.

1.2 Report Outline

Please note that all images of block designs and simulations have been included as pictures in the attachments.

Chapter 2: A presentation of AXI-Stream and all the cores used for communication. AXI-Stream is used to send data from memory to the video scaler and to send data from the video scaler to memory or to the video subsystem. The chapter also contains the block design created for this project and an example design where the video scaler was implemented.

Chapter 3: A presentation of the video subsystem, which is responsible for sending video to the HDMI/VGA-port. The chapter contains information about all the cores that are present in the subsystem and a block design of the video subsystem.

Chapter 4: A presentation of the video scalers, as well as their visual results. The chapter contains several images that have been scaled up or down with some variations in how the algorithm is implemented. The chapter also includes the theory behind the video scalers and how it was approached.

Chapter 5: Discussion about the results, in terms of utilization and timing when implementing the scaler in larger system. The chapter also includes results of improvements made to the downscaler and some suggestions for future work.

Chapter 6: Conclusion of the thesis.

Appendix A.1: The code for the implemented downscaler.

Appendix A.2: The code for the simulated downscaler.

Appendix B: The code for the simulated upscaler.

Chapter 2

Communication

2.1 Advanced eXtensible Interface

In this project, the Advanced eXtensible Interface (AXI) is used to communicate. AXI is a standardized IP interface protocol based on the Advanced Microcontroller Bus Architecture (AMBA) specification. AXI is suited for designs that requires high bandwidth and low latency. There are three types of AXI4 interfaces:

- AXI4 - For high-performance memory-mapped requirements
- AXI4-Lite - For simple, low-throughput memory-mapped communication
- AXI4-Stream - For high-speed streaming data

The benefits of AXI4 is increased productivity, flexibility and availability. The productivity is increased by standardizing the AXI interface, since all communication cores provided by Xilinx follow the same protocol.

The protocol is flexible by providing three different versions based on what is needed. In this thesis it is AXI-Lite for control signals and AXI-Stream for data. Both AXI4 and

AXI4-Lite consists of five different channels:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

Data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary. AXI4 allows burst transaction up to 256 data transfers, while AXI4-Lite allows only 1 data transfer per transaction. To achieve very high data throughput, AXI4-compliant systems have some features, in addition to bursting, like data up sizing and downsizing, multiple outstanding addresses, and out-of-order transaction processing. It also allows different clocks for each master-slave pair and insertion of register slices to aid in timing closure, AXI-Guide-Xilinx [2].

2.2 AXI4-Stream

The AXI4-Stream protocol defines a single channel for transmission of streaming data. The channel is modeled after the Write Data channel of AXI4, but it can burst an unlimited amount of data.

AXI4-Stream is used for applications that typically focus on data-flow where the concept of an address is not present or not required. Each AXI4-Stream acts as a single unidirectional channel for a handshake data flow. A DMA (Direct Memory Access) or VDMA (Video Direct Memory Access) can be used to move streams in and out of memory.

Figure 2.5 shows both inputs and outputs of Master AXI-Stream and Slave AXI-Stream. All the AXI-Stream master signals is outputted from the AXI-peripheral, except for m_axis_tready, which is an input signal.

The $READY$ signal is used together with the $VALID$ signal to create a handshake. The handshake is used to transfer data and control information. The master will send a $VALID$ signal to indicate that a transfer of control information or data is available. The slave accepts the data or control information by asserting the $READY$ signal back to the master. A transfer will only occur when both $VALID$ and $READY$ are $HIGH$.

There are three ways to create a handshake as shown in figure 2.2 and table 2.1.

Signal order	Reaction
Both $VALID$ and $READY$ are set HIGH	Immediate transfer
$VALID$ is set HIGH before $READY$	Transfer when slave asserts $READY$, master is waiting
$READY$ is set HIGH before $VALID$	Transfer when master sends $VALID$, slave is ready

Table 2.1: Handshake

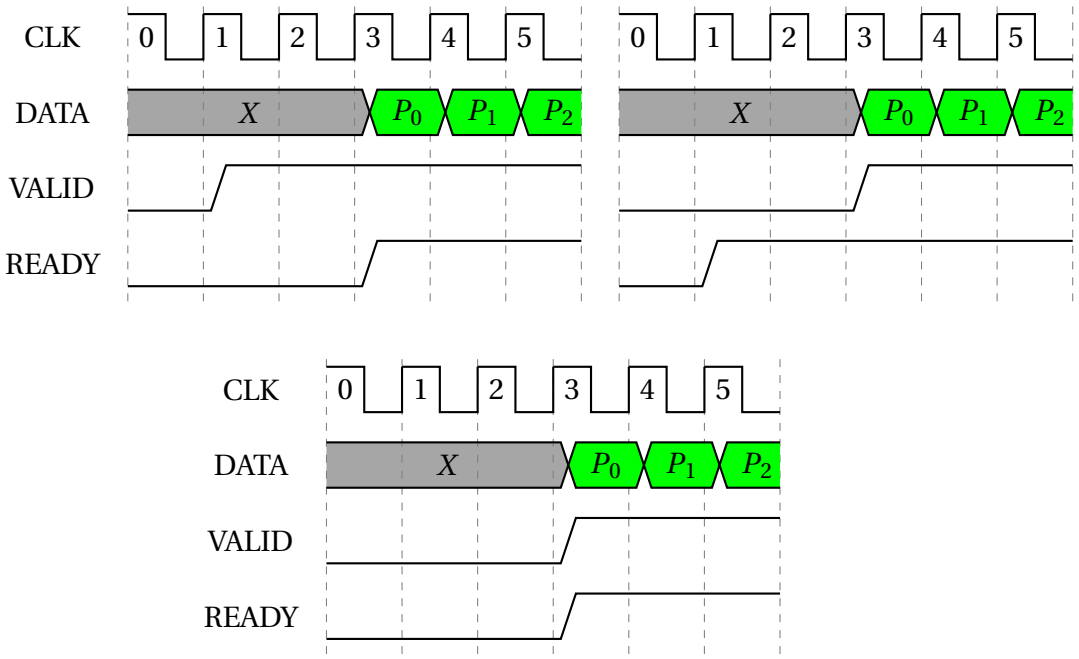

Figure 2.1: All types of valid handshake where VALID and READY is set in different order

2.3 AXI Interconnect

The AXI Interconnect core IP connects on or more AXI memory-mapped master devices to one or more memory-mapped slave devices. In the project it is used to connect the master (e.g. processor, DMA/VDMA) and slave (e.g. memory, HDMI). The interconnect multiplexes and demultiplexes data and control information between connected masters and slaves and it ensures which bus master is allowed to initiate data transfers.

The AXI Interconnect core allows any mixture of master and slave devices to be connected to it, which can vary from one another in terms of data width, clock domain and AXI sub-protocol (AXI4-Stream, AXI4 or AXI4-Lite).

When the interface characteristics of any connected master or slave device differ from those of the crossbar switch inside the interconnect, the appropriate infrastructure cores are automatically inferred and connected within the interconnect to perform the necessary conversions. Interconnect is general-purpose, and is typically deployed in all systems using AXI memory-mapped transfers, AXI-Interconnect-Xilinx [3].

2.4 AXI-Stream FIFO

The AXI4-Stream FIFO core allows memory mapped access to AXI4-Stream interface. The IP is easily manageable as the AXI4-Stream interfaces are transparent and no extra signaling is needed. The FIFO have configurable data width of 32, 64, 128, 256 or 512 bits and for this project 32 bits were used as the data width needed is 8 bits, AXI-FIFO-Xilinx [1].

The depth of the FIFO is between 512 to 128 000 locations. The image with the greatest width in this project had a width of 640 pixels and a height of 480 pixels. Since each of the pixels hold three different values, each representing red, green or blue, the image will require 892 800 locations if the whole image should be stored temporary, but this is neither possible (unless multiple FIFOs are used) nor necessary.

The maximum number of locations needed in the FIFO with the current design is approximately 960 (320 pixels), which is the number cycles the S_AXIS_TDATA will be suspended while the video scaler is writing to the M_AXIS_TDATA. The scaler will suspend reading from S_AXIS_TDATA when writing to the AXI4-Stream to avoid errors. The equation for the FIFO size for a downscaling module is:

$$\frac{3(RGB) \times WIDTH}{2} \tag{2.1}$$

For the upscaling module, the number of locations needed in the FIFO is eighth times greater than the downscaling module for an image with the same size. As seen in figure 2.4, the scaled up version contains 16 times the amount of pixels that the scaled down version, and four times the amount of original image. This results in a larger load of data in the FIFO, which can be calculated by:

$$2(Rows) \times 2 \times 3(RGB) \times WIDTH \tag{2.2}$$

The equation is based on the fact that the upscaling module will write two rows of pixels for every row it reads as well as each of the rows being twice as long as the original. For the image with a width of 640 pixels, the FIFO will need at least 7680 locations. The FIFO depth is therefore set to 8192 locations.

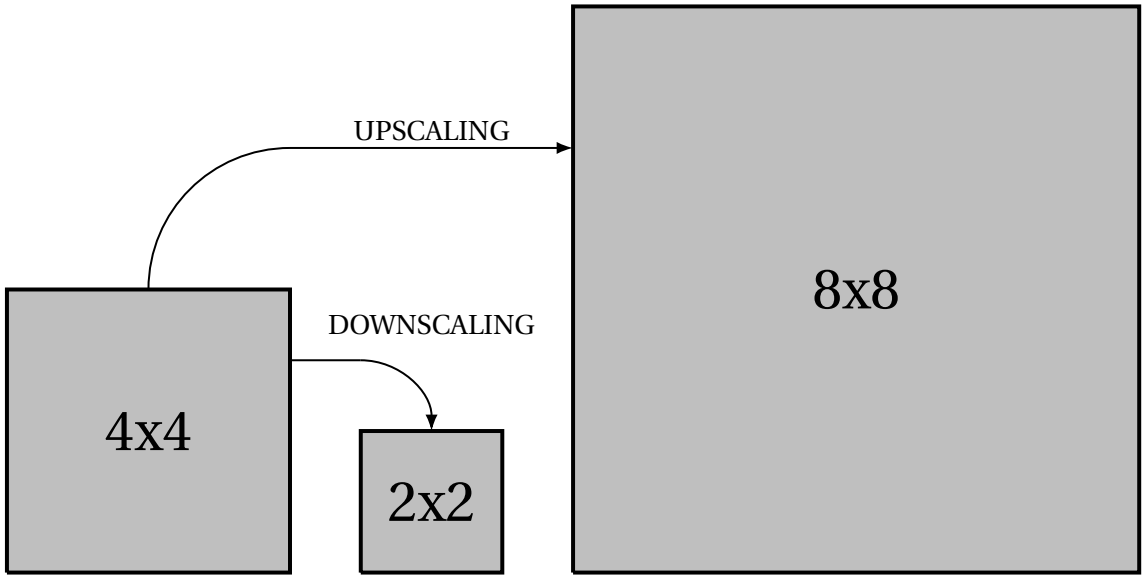

Figure 2.2: Different sizes of original image, downscaled image and upscaled image

2.5 AXI DMA - Direct Memory Access

The Xilinx AXI Direct Memory Access (AXI DMA) core is a soft Xilinx IP core. It provides high bandwidth direct memory access between memory and AXI4-Stream target peripherals. It supports AXI4-Stream data width of 8, 16, 32, 64, 128, 256 and 1024 bits. Primary high-speed DMA data movement between system memory and stream target is through the AXI4 Read Master to AXI4 memory mapped to stream (MM2S) Master, and AXI-Stream to to memory-mapped (S2MM) Slave to AXI4 Write Master. AXI DMA also enables up to 16 multiple channels of data movement on both MM2S and S2MM paths in scatter/gather mode. The AXI DMA provides the ability to queue multiple transfer requests using nearly the full bandwidth capabilities of the AXI4-Stream buses.

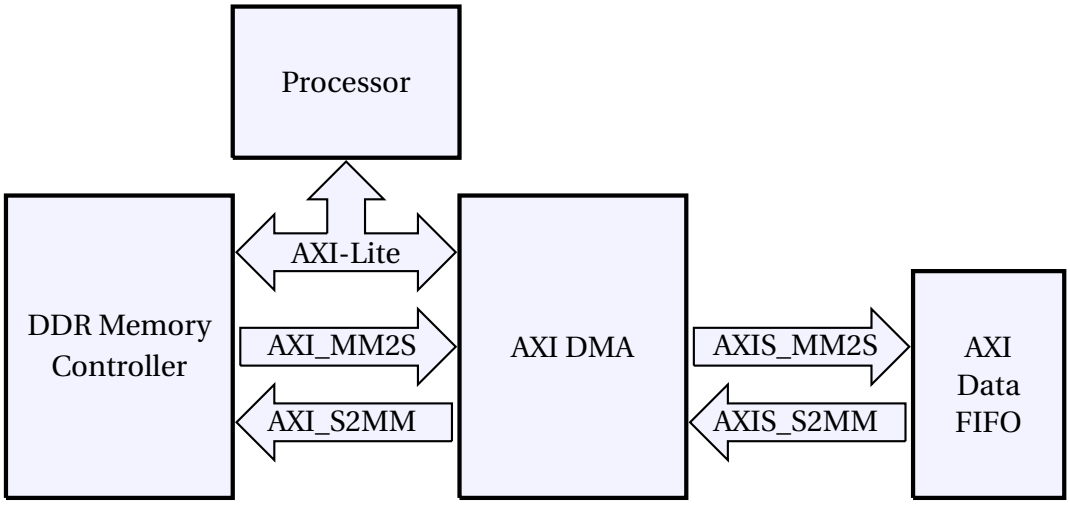

Figure 2.3: Block Diagram of AXI DMA system, provided by FPGA-Developer [7]

Figure 2.5 show a design provided by FPGA-Developer. In this design, the processor and DDR memory controller are contained within the Zynq's processing system (PS), while the DMA and AXI Data FIFO are implemented in the Zynq's programmable logic (PL). The AXI-lite bus allows the processor to communicate with the AXI DMA to setup, initiate and monitor data transfers. The $AXIS\_MM2S$ and $AXIS\_S2MM$ are memory-mapped AXI4 buses and provide the DMA access to the DDR memory. The $AXIS\_MM2S$ and $AXIS\_S2MM$ are AXI4-streaming buses, which source and sink a continuous stream of data, without addresses, DMA-Xilinx [6].

2.6 AXI VDMA - Video Direct Memory Access

The Video Direct Memory Access (VDMA) provides high-bandwidth direct memory access between memory and AXI4-Stream video type target peripherals. It is designed to allow for efficient high-bandwidth access between the AXI4-Stream video interface and the AXI4 interface.

The VDMA is suited for 2D-transfers such as video/image-transfer. Figure 2.4 is a block diagram of the VDMA core. The "control and status"-block receives commands based on how the registers are programmed and respond accordingly with READ/WRITE commands to the DataMover, VDMA-Xilinx [13].

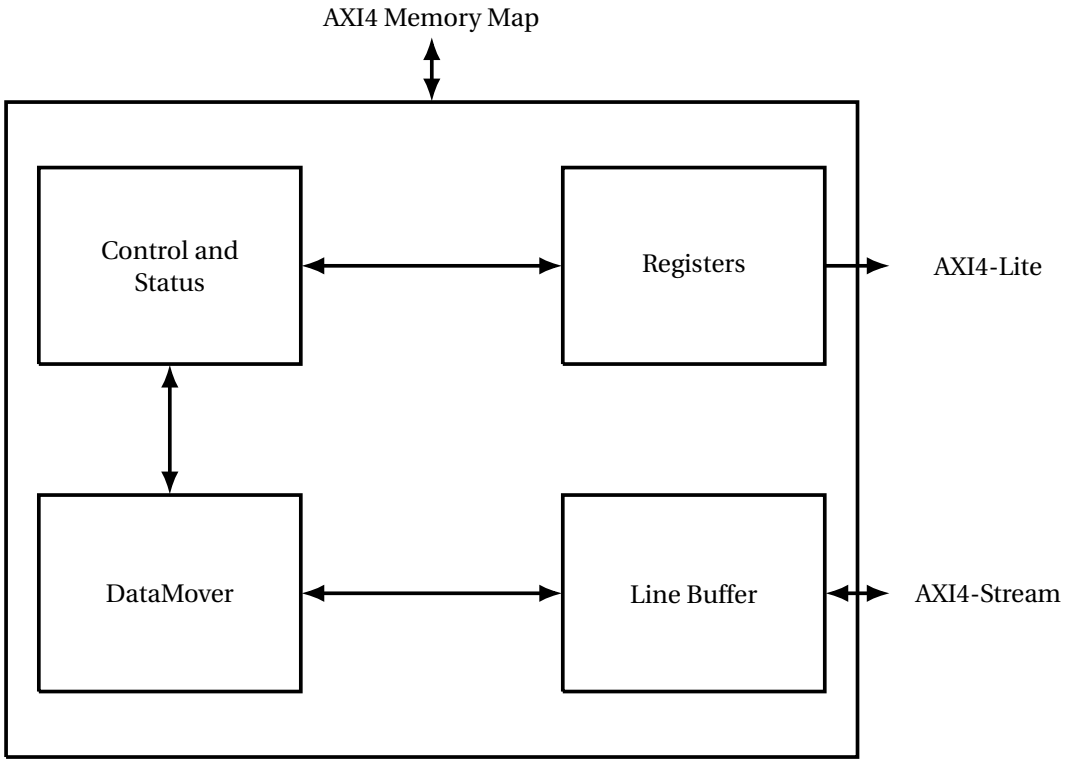

Figure 2.4: AXI VDMA Block Diagram

2.7 System

The complete video scaling system will consist of three large modules, the video module, (which contains the video scaler and the all the IP cores needed to transfer data to a HDMI-port,), the video scaler module, and the communication module. The communication module consists of a processor module and a memory module, along with all the IP cores needed to transfer data between memory and the video scaler. AXI4-Stream and AXI4-Lite will be the means of communication between the modules.

Figure 2.5 is a custom made AXI4-Stream generator, which developed from Vivado's $IP\ Packager$. To initiate it, press "Tools", then "Create and Package New IP", then "Create a new AXI4 peripheral". When you have created an AXI4 peripheral, add both an AXI4-Stream Slave and an AXI4-Stream Master, to provide the correct ports in and out of the custom IP. The IP can then be edited to create a AXI4-Stream generator. The most important signals in the AXI-Stream generator is $DATA$, $VALID$ and $READY$, as all of them are crucial for the AXI-Stream generator to send and receive data.

In the complete version, the AXI-Stream generator was replaced with the scaling module as the generator was meant to create traffic to ensure that the communication between the module worked as planned. In the AXI-Stream generator, the $LAST$ signal was low until a burst of 8 transfers has occurred. The number of transfers in one burst could be changed, but 8 was the default when creating an AXI peripheral. For the scaling module, the $LAST$ signal was set high when the scaling module had completed writing a row to the M_AXIS_TDATA, since it would not send more data until two more rows had been read through the S-channel.

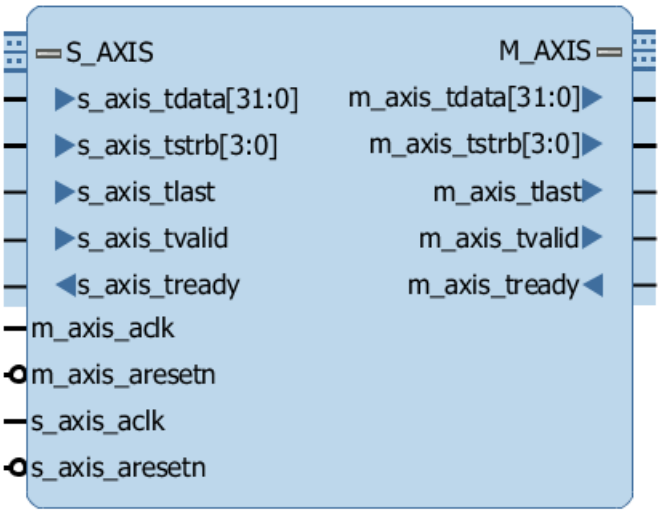

Figure 2.5: AXI4-Stream inputs and outputs for both Master and Slave

Figure 2.6 shows a block diagram of the system where the AXI-Stream generator is generating traffic by reading and writing to memory, through the DMA and MIG (Memory Interface Generator). The system was intended to be a simple system which could both read and write through the use of the AXI-Stream generator before adding more cores to the system.

Figure 2.7 illustrates the communication of the system, excluding the video module, as the modules have not been able to work together in this project. The block diagram is based upon the block design made in Vivado, which is shown in figure 2.8. The block design has a lot of similarity to the processor subsystem of the AXI VDMA Reference Design, XAPP742 from Xilinx [17]. The AXI VDMA Reference Design contains a video module, showing which cores are needed to get video out by HDMI. Since the design from this thesis does not have a functional video-out subsystem, the data have been read to the scaling module, scaled and sent back.

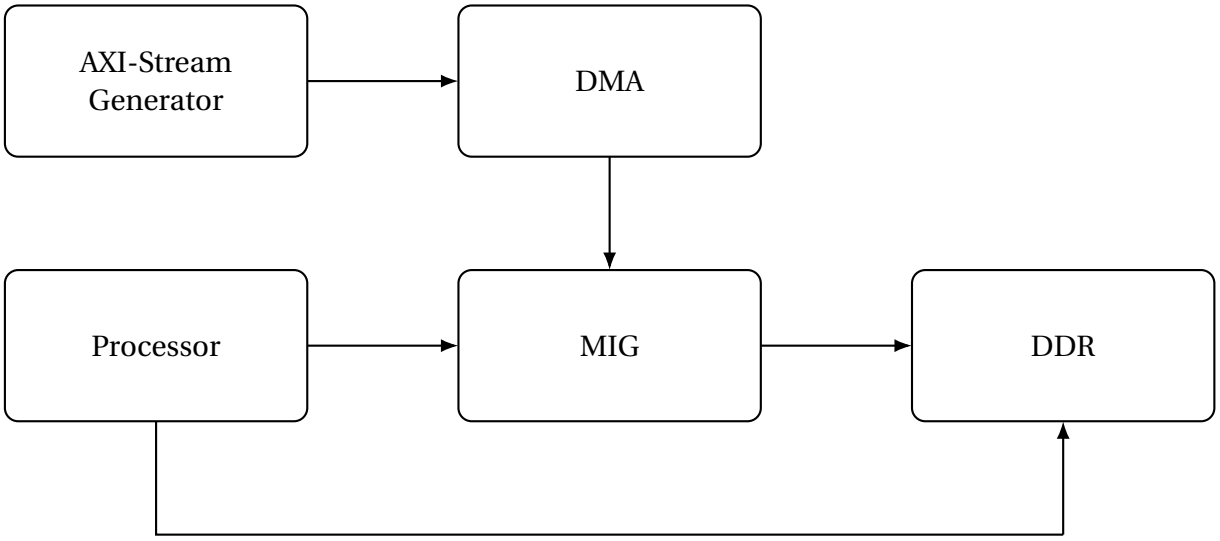

Figure 2.6: Block Diagram of communcation between AXI-Stream Generator and memory

The block design in figure 2.8 is made for writing to memory, since the AXI-Stream generator does not have any inputs on the AXI-slave interface. The changes needed to make the generator both read and write from memory are easy to implement, but the system was made this way to establish one-way communication. Apart from this and the lack of a AXI-Stream FIFO, the system contains all the cores needed to communicate with memory. The module does also contain an AXI GPIO core, which is a general purpose input/output interface for AXI-Lite that can be used to control external devices or access internal properties.

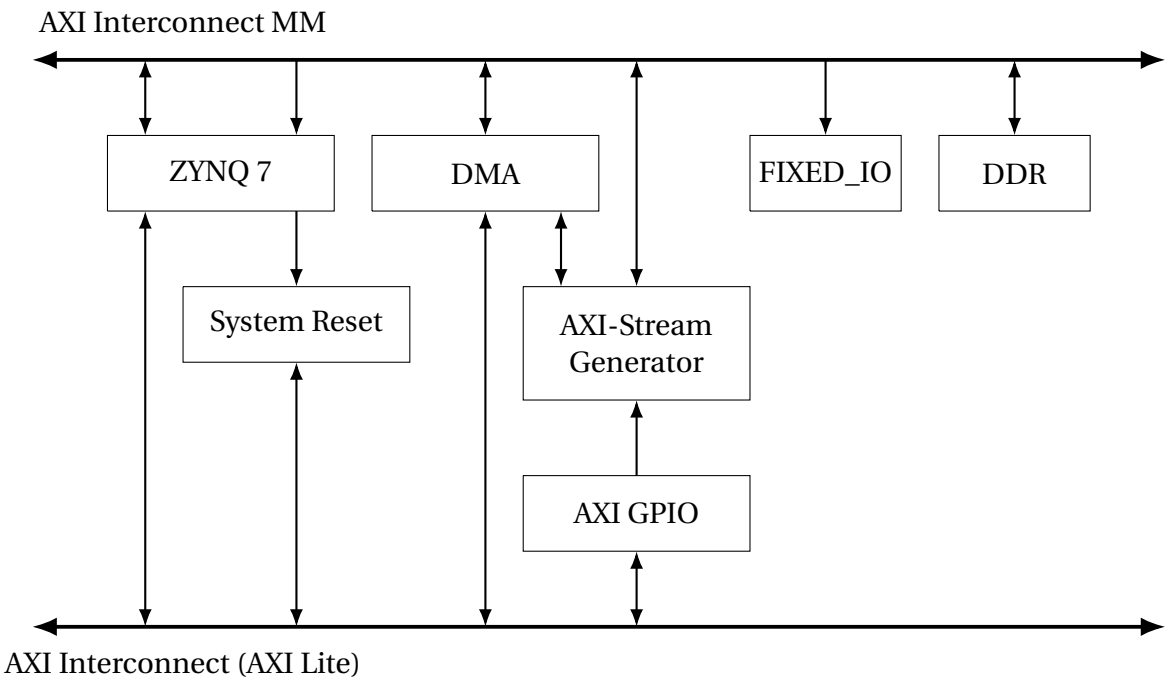

Figure 2.7: Block diagram of a AXI System with a ZYNQ 7 Processor

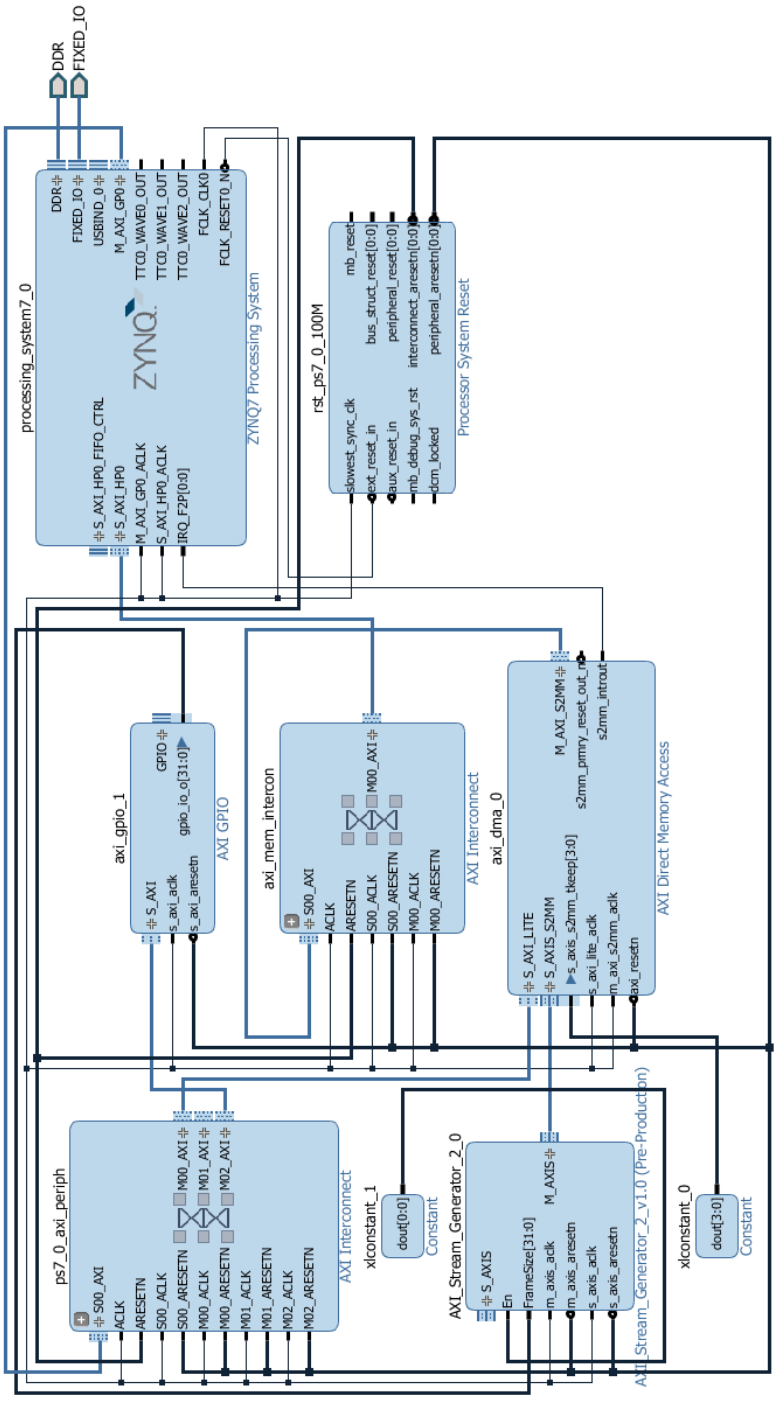

Figure 2.8: Block Design of AXI Stream System made in Vivado

2.8 DMA Interrupt example

Since the communication between the control module and the video module did not work as intended, an example provided by Xilinx-DMA-ex [19] was used to test the video scaler on the FPGA. The system in figure 2.9 shows a lot of similarities to figure 2.8.

The block design of figure 2.10 and figure 2.9 are both part of the same block design. The signals M_AXI_MM2S and M_AXI_S2MM, from figure 2.10 are outputs of the AXI DMA, which are the input signals of AXI Interconect 2 from figure 2.9. The input signals are named $S\_AXI\_DMA\_MM2S$ and $S\_AXI\_DMA\_S2MM$. The use of S as the initial letter is to indicate a $slave$-port or input-port, though the slave-ports have output-pins such as the s_axis_tready, from the downscale module in figure 2.10. Likewise, the M indicates $master$-port which is an output-port. The signals $mm2s\_introut[0:0]$ and $s2mm\_introut[0:0]$ from figure 2.10 are also connected to the signals $dma\_mm2s\_irq[0:0]$ and $dma\_s2mm\_irq[0:0]$ from figure 2.9. The system is designed to send an interrupt-signal to the processor system for each transaction, which makes the system wait for the next input to the system. The DDR and $FIXED\_IO$ signals are the only signals that can be reached by SDK, which is used to test the system. To read from the FPGA, a PuTTY-terminal, which is a serial console, was used on the UART-pin J14 with the speed set to 115200.

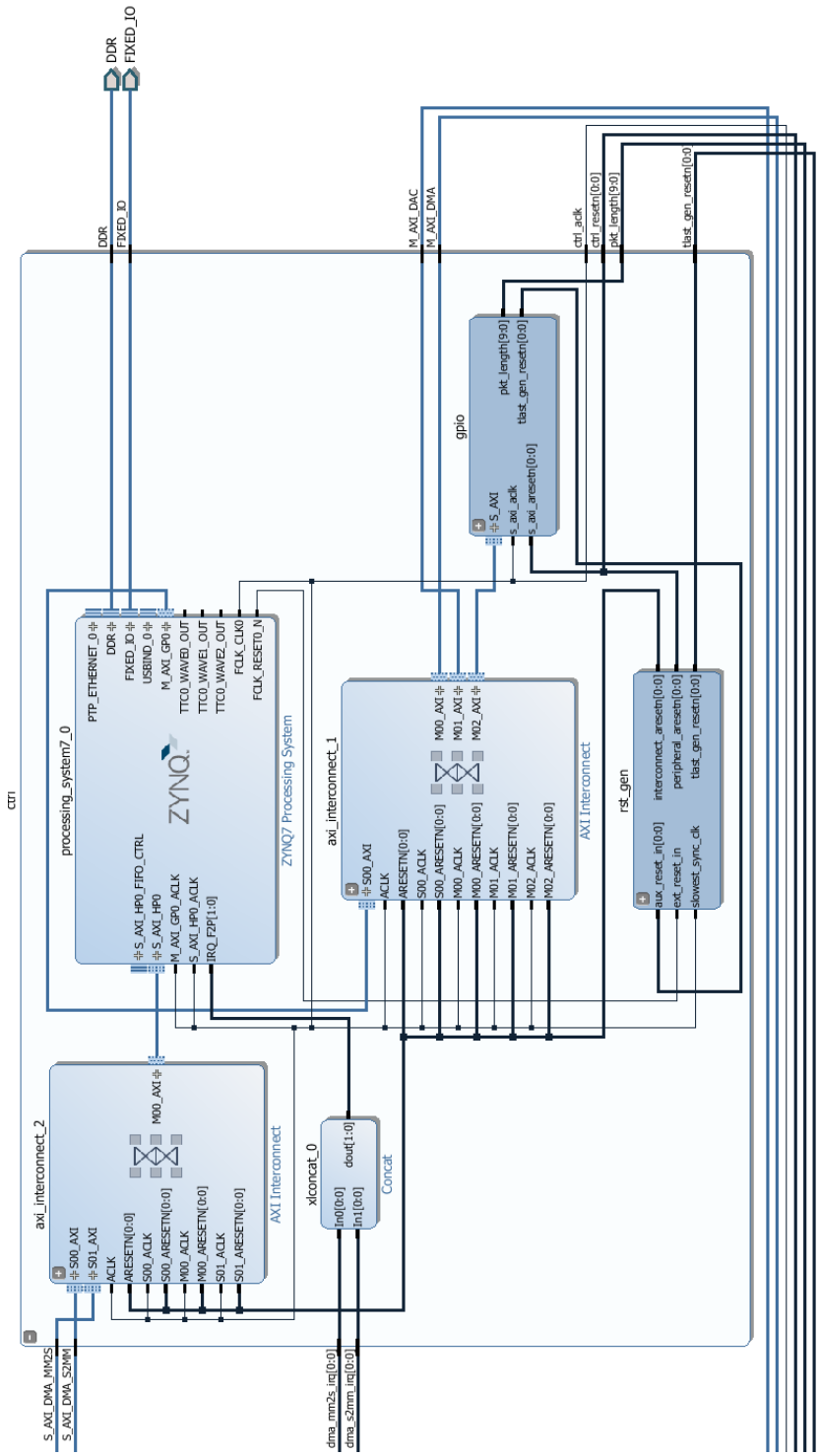

Figure 2.9: Control part of AXI4-Stream system with downscaler

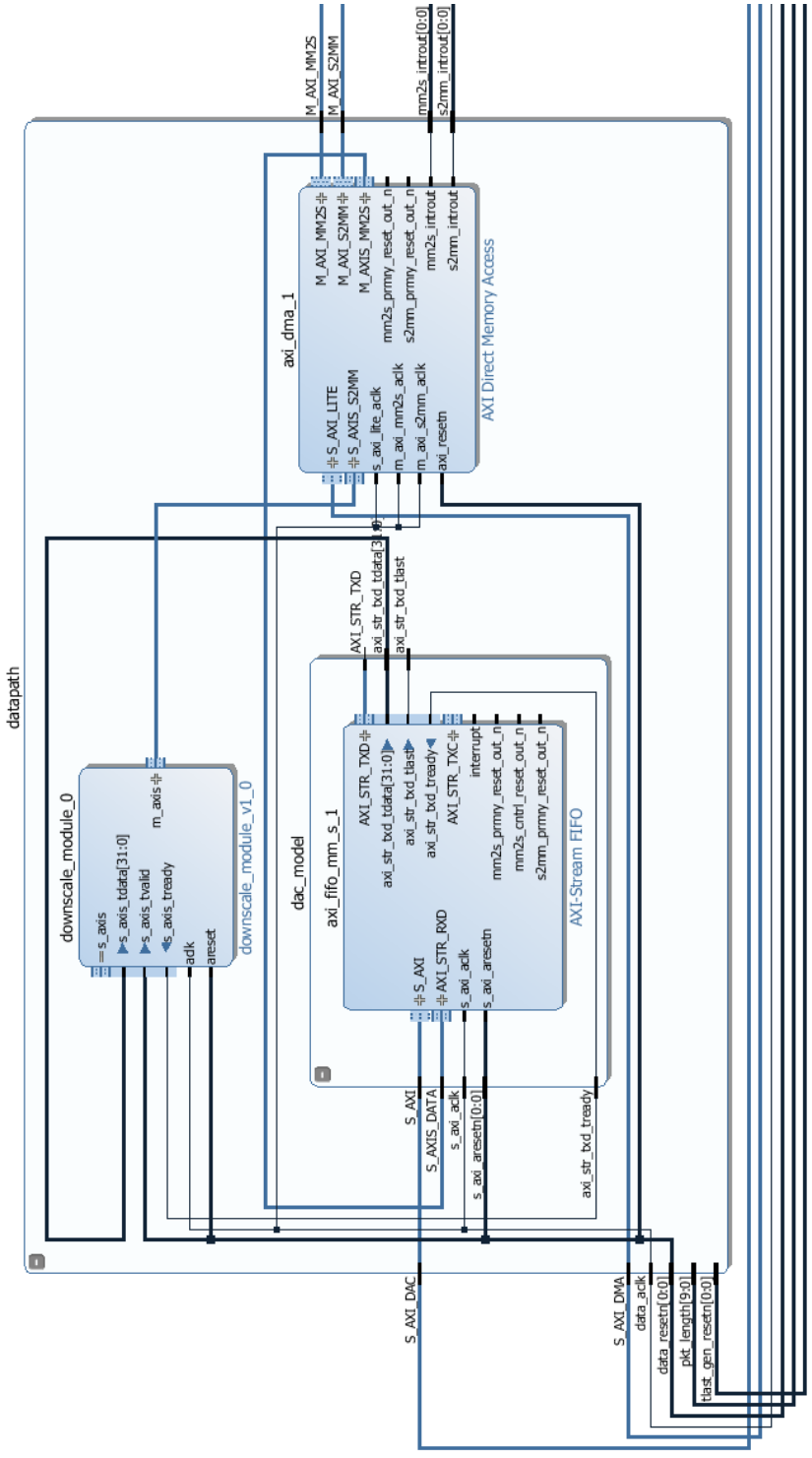

Figure 2.10: Datapath part of AXI4-Stream system with downscaler

Chapter 3

Video Module

The video subsystem is intended to display the scaled video, or in this case, images, which has been read by the memory module and rescaled by the video scaler. The scaled video was intended to be sent through the video module and out by HDMI. Figure 3.1 is the block design of the video subsystem created with Vivado IP Integrator.

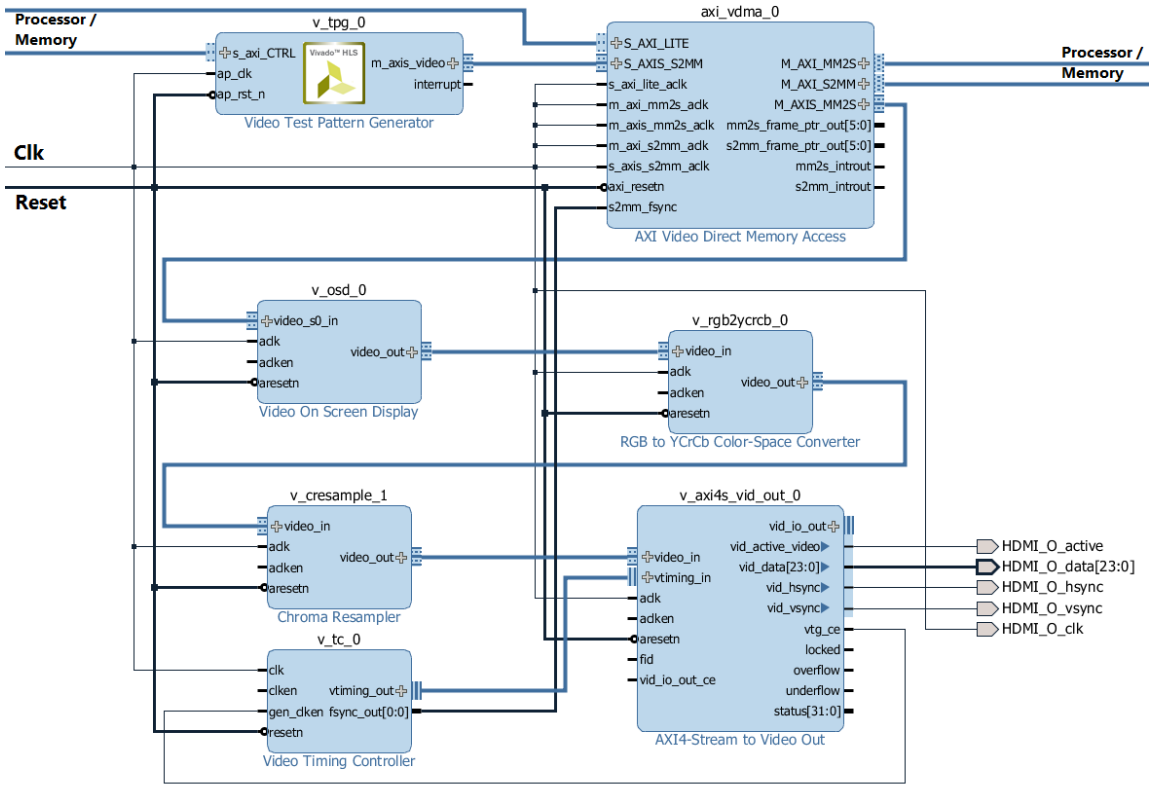

Figure 3.1: Block Design of the video module, designed from Xilinx [17]

3.1 Video Test Pattern Generator

The Video Test Pattern Generator generates test patterns which can be used when developing video processing cores or bringing up a video system. The test patterns can be used to evaluate and debug color, quality, edge, and motion performance, debug and assess video system color, quality, edge, and motion performance of a system, or stress the video processing to ensure proper functionality.

The Video Test Pattern Generator core supports bidirectional data throttling between its AXI4-Stream Slave and Master interfaces. The core uses handshake, which is mentioned in chapter 5, meaning that if the slave side data source is not providing valid data samples, the core cannot produce valid output samples after its internal buffers are depleted, VTPG-Xilinx [16].

3.2 Video On Screen Display

The Video On Screen Display core is used for alpha blending, composition and simple text and graphics generation. Alpha blending indicate the degree of opaqueness in a picture, or how translucent a color is in the picture. In figure 3.2, there are three layers, where each layer resides closer or further from the observer. The order in the figure is green in the back, red in the middle and blue in the front, which can be observed at the left figure with no transparency. The right figure contains the same colors, in the same order, but due to opacity at 50%, the edges around all of the circles are visible, and the combination of different colors give each overlapping area a new shade. The red figure below shows a black filter with increasing opacity from left to right.

The Video On-Screen Display produces output video from multiple external video sources and multiple internal graphics controllers, where each video and graphics source is assigned an image layer that can be dynamically positioned, resized, brought forward or backward, and combined using alpha-blending. It supports AXI4-Stream Video Protocol on the input interfaces, allowing easy integration with other Video IP cores such as AXI VDMA, Video Scaler, Color Space Converters, Chroma Resampler and Video Timing Controller, VOSD-Xilinx [14].

The maximum throughput of the Video On-Screen Display can be calculated by:

$$\frac{cycles\ per\ second \times lines\ per\ frame \times channels\ per\ pixel \times bits\ per\ channel}{cycles\ per\ frame} \tag{3.1}$$

The maximum output of the AXI4-Stream can be calculated by:

$$\frac{cycles\ per\ second \times 4096 \times channels\ per\ pixel \times bits\ per\ channel}{4097} \tag{3.2}$$

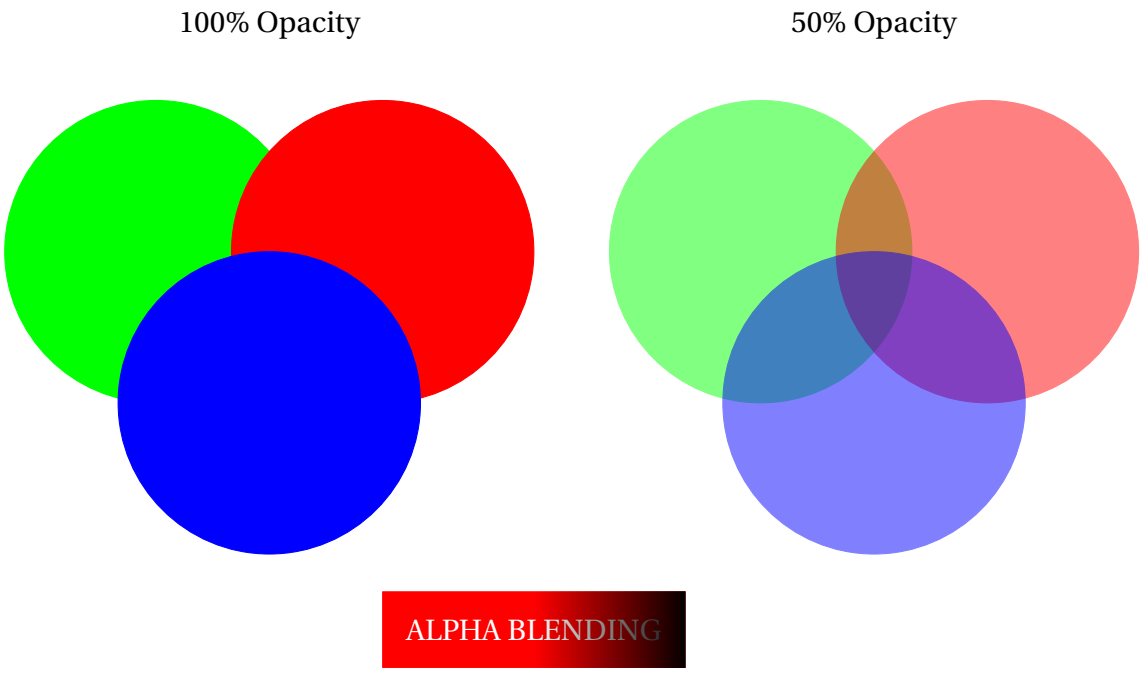

Figure 3.2: Variation in opacity

3.3 RGB to YCrCb Color-Space Converter

The RGB to YCrCB Color-Space Converter IP is provided by Xilinx, and it is set by default to 1920 pixels per scanline and 1080 scanlines per frame, but both of these values can be set anywhere between 32 and 7680, which is needed as since the default values does not fit any of the images in this thesis. The converter use the SD ITU 601 or Rec. 601, which is a recommended standard for encoding interlaced analog video signals in digital video form. It uses a color encoding scheme which is known as Y'CbCr 4:2:2. Y'CbCr consists of the Y' which luma and it represents the brightness or the black-and-white portion in an image. Cb is chroma-blue and Cr is chroma-red, which represents the color information needed to add colors to a monochrome image. The luma and the chroma are separated because human vision has finer sensitivity to luminance differences than chromatic differences, giving video systems the opportunity to have lower resolution on chromatic information than the luminance information, RGB2YCBCR-Xilinx [12]. The basic equations for finding the conversion is:

$$Y = R \times 0.257 + G \times 0.504 + B \times 0.098 + 16 \tag{3.3}$$

$$Cb = -R \times 0.148 - G \times 0.291 + B \times 0.439 + 128 \tag{3.4}$$

$$Cr = R \times 0.439 - G \times 0.368 - B \times 0.071 + 128 \tag{3.5}$$

Table 3.1 shows the RGB values of all the colors named in figure 3.3 where all the colors are a product of red, green and blue. All colors have three values between 0 and 255 (8 bits), where darker shades of color have lower values and lighter shades have higher values.

Black has the value (0, 0, 0) and is therefore origo in the system, while white has the highest value (255, 255, 255). As the name RGB indicates, RGB, (255, 0, 0) are red, (0, 255, 0) are green and (0, 0, 255) are blue, and all other colors are a mixture of these three.

	Black	Red	Green	Yellow	Blue	Magneta	Cyan	White
R	0	255	0	255	0	255	0	255
G	0	0	255	255	0	0	255	255
B	0	0	0	0	255	255	255	255

Table 3.1: RGB Color bars

When converting the colors of table 3.1 to Y'CbCr using the equations (6.3), (6.4) and (6.5), the result will be as presented in table 3.2, where 16 is the lowest possible value, and 240 is the highest possible value. Note that black and white, which were opposite in all aspects of RGB got the same Cb and Cr values, making Y' or brightness the difference between them.

	Black	Red	Green	Yellow	Blue	Magneta	Cyan	White
Y'	16	82	145	210	41	107	177	235
Cb	128	90	54	16	240	202	166	128
Cr	128	240	34	146	110	221	16	128

Table 3.2: Y'CbCr Color bars

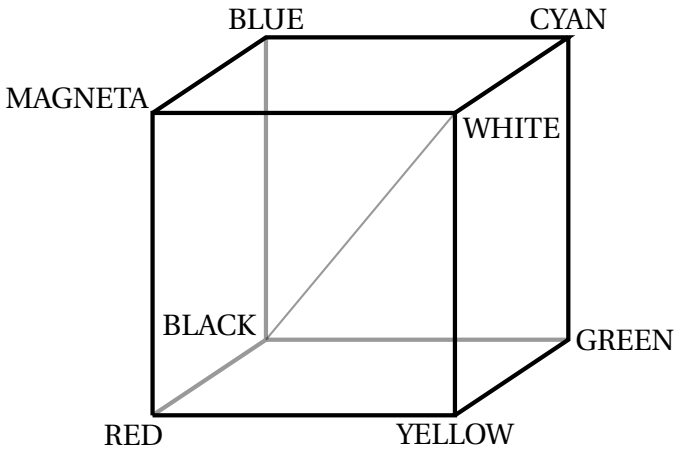

Figure 3.3: RGB represented as three dimensional space

3.4 Chroma Resampler

The human eye is not as receptive to details regarding color details as it is to brightness details, which is why using the YCbCr color-space have some benefits compared to RGB. When using YCbCr, the Y-layer (Luma-layer or brightness) will be sampled at the same speed as all layers in RGB would have done, but the Cb (Chroma Blue) and Cr (Chroma Red) can be sampled at a lower rate as the changes are less visible.

The Chroma Resampler LogiCORE IP converts video data between commonly used chroma formats. The supported formats are 4:4:4, 4:2:2 and 4:2:0. The conversion requires fast VRAM calculations on 2x2 pixels for compression using a sliding average on each pixel grouped in 2x2, resampler Xilinx [11].

When converting from YUV4:4:4 to YUV4:2:2 the following reduction would be implemented:

Y: Luma, U: Chroma, V: Chroma.

For YUV4:4:4:

There are 3 octet (one byte) values for each pixel in YCbCr.

$4 \times 8 + 4 \times 8 + 4 \times 8 = 96\ bits\ per\ 4\ pixels = 24\ bits\ per\ pixel$

For YUV4:2:2: For a 2x2 group of pixels there are 4 Y samples and 2 U and 2 V samples each.

$4 \times 8 + 2 \times 8 + 2 \times 8 = 64\ bits\ per\ 4\ pixels = 16\ bits\ per\ pixel$

This resampling will reduce the number of bits per pixel sent to HDMI by 1/3.

3.5 Video Timing Controller

The Video Timing Controller (VTC) core is a general purpose video timing generator and detector. The core is highly programmable and it provides easy integration into a processor system for in-system control. It also contains an optional AXI4-Lite interface and it supports video frame sizes up to 8192 x 8192 pixels. The IP can automatically detect and generate horizontal and vertical video timing signals, which is useful for managing the synchronize processes, VTC-Xilinx [15].

3.6 AXI4-Stream to Video Out

The AXI4-Stream to Video Out core is the interface from AXI4-Stream interface to a video source such as HDMI. The core works with the Video Timing Controller core, which provides the video timing generation when the AXI4-Stream to Video Out core is in slave mode. If it is in master mode, it will automatically synchronize AXI4-Stream Video to video timing. The IP is able to handle asynchronous clock boundry crossing between AXI4-Stream clock domain and video clock domain and it contains a FIFO depth from 64 to 8192 locations with input width of 8-256 bits, AXI2VIDEO-Xilinx [4].

Chapter 4

Video Scaler

A video scaler is used to convert signals from one display resolution to another. The most common use is to convert signals from a lower (e.g. 480p) resolution to a higher resolution (e.g. 1080p), but it is also used to downscale the resolution of the video signals, to fit on smaller displays such as a mobile phone or a tablet. A video scaler is often combined with other video processing devices or algorithms to create a video processor that improves the definition of the video signals. To create a scaler that scales up an image, the first step was to expand every pixel, which in this case was by 2x2 pixels (see figure 4.6), creating a new image with four times the size of the original picture. In the first version, each pixel was multiplied both horisontally and vertically, where all the pixels had the same color as the original pixel. The next step was to create the scale2x-algorithm, which compares neighboring pixels to decide the color of the new pixels, though the results did not impress as the algorithm was not able to recognize patterns. Figure 4 is a block diagram, showing how the video scaler will fit into the AXI4 system.

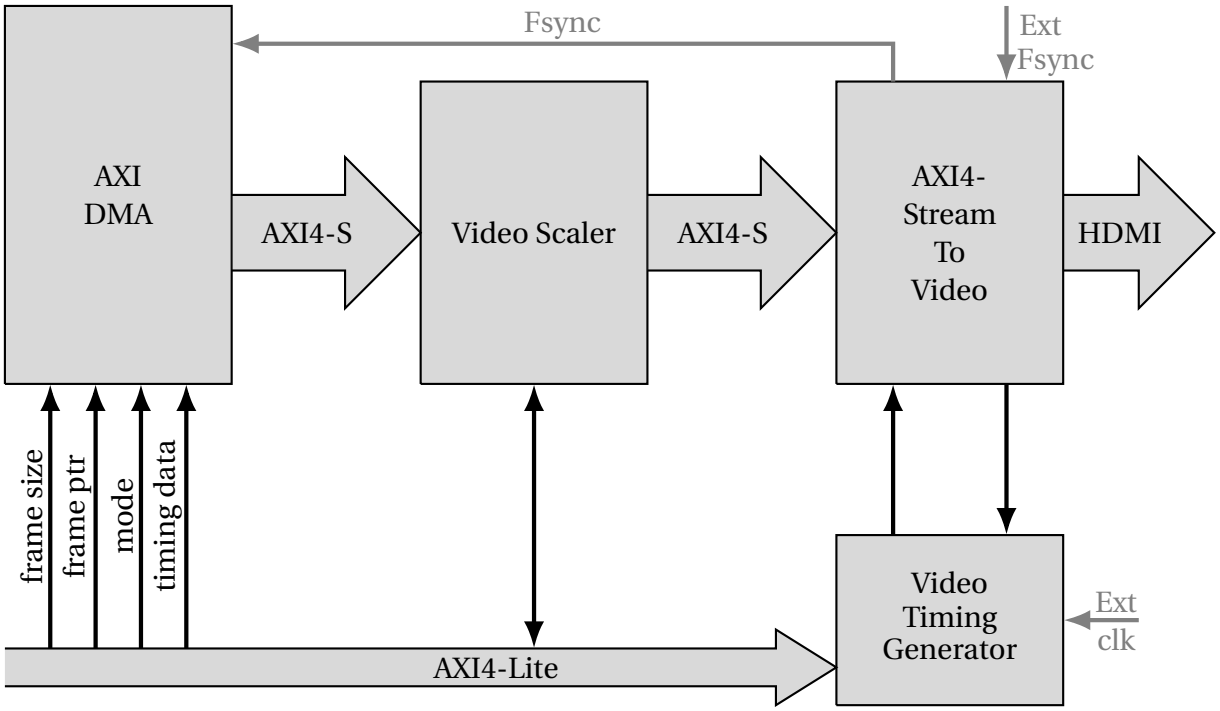

Figure 4.1: Location of Video Scaler

4.1 Data to Vivado

In order to read data from an image, a conversion is needed as Vivado does not contain a library that supports reading Portable Network Graphics (PNG) files. The converted data can either be stored in a text-file or COE-file. The COE-file can be directly put into either Block RAM or Block ROM before running the program, making it easier to use when implemented on the FPGA.

The BRAM does however have limited space as each BRAM have a size of 36K bit each.

This would not be enough for either of the pre-scaled images as even the smallest of the images exceeds the 36K bit limit.

To instantiate the COE file the two keywords in table 4.1 is used. Memory_initialization_radix can be set to 2 for bit values, 10 for decimal values or 16 for hexadecimal values. Memory_initialization_radix contains all the data which should be preinserted into the BRAM in whatever format is set by the initializated radix.

Table 4.1: COE keywords

Keyword	Description
MEMORY_INITIALIZATION_RADIX	Used for memory initialization values to specify the radix used.
MEMORY_INITIALIZATION_VECTOR	Used for block and distributed memories.

To put the COE-file into the BRAM, press IP Catalog as shown in figure 4.2 and load it into the RAM as shown in figure 4.3.

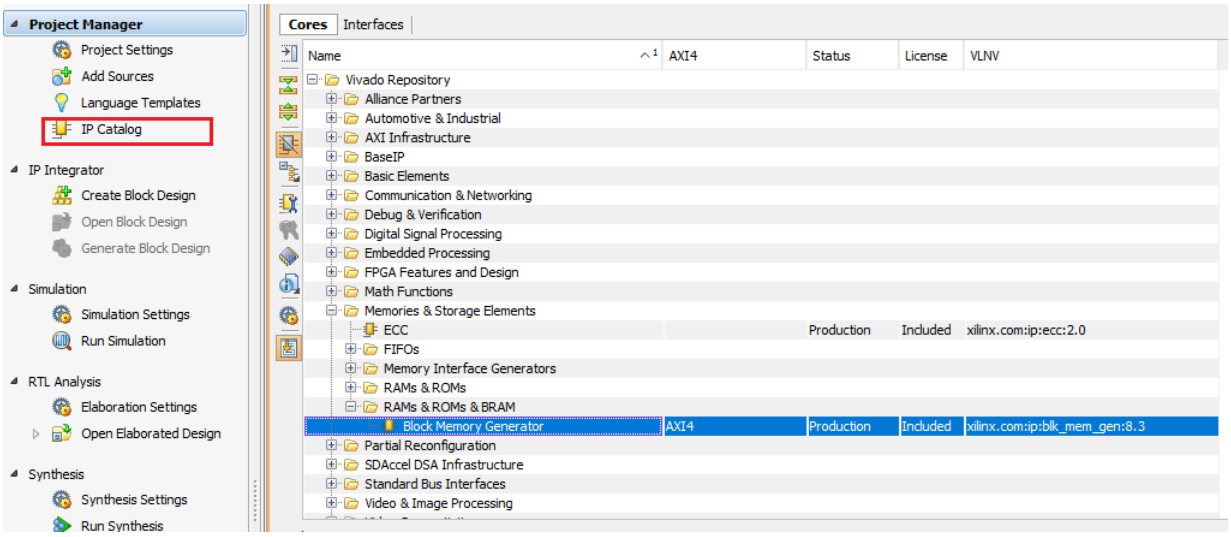

Figure 4.2: Adding RAM to the design

The following calculations are meant to illustrate the size of the images compared to limits of the BRAM. For the image in figure 4.4, with 128x64 pixels, the minimum number of bits needed is:

$$8\ bits \times 3 \times 128 \times 64 = 196\,608\ bits \tag{4.1}$$

This is over five times the amount which can be stored in a single BRAM, which means the image must be divided among at least six BRAMs in order to fit, if the BRAM had 8 bit data width.

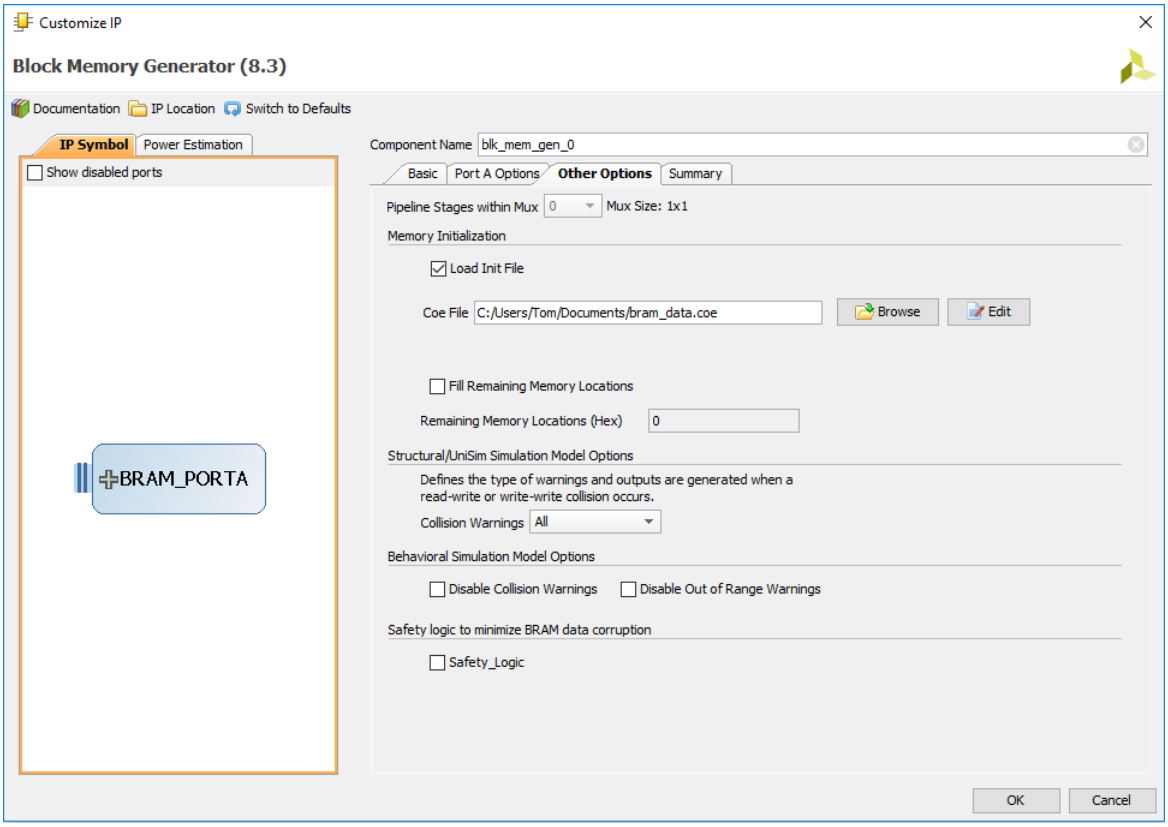

Figure 4.3: Loading data into BRAM

The different data widths of the BRAM is 3, 6, 12, 24, 48, 96 and 192, making 12 bit width the preferred data width. Another approach is to store the three different 8-bits values in a 24 bit width data slot, but it has not been done in this project.

For the scaled up version with 256x128 pixels, the result is 786 432 bits, and for the scaled down version with 64x32 pixels, the result is 49 152 bits. As for the picture in figure 4.7 the number of pixels are 480x480 making the minimum number of bits needed:

$$8\ bits \times 3 \times 480 \times 480 = 5\ 529\ 600\ bits \tag{4.2}$$

The scaled up version of the picture, which is 960x960 pixels, will require 22 118 400 bits, which would require 615 BRAMs to store all the data. The scaled down version will require at least 1 382 400 bits.

The BRAM may not be suited for large amounts of data, but it would be possible to use it to store data locally which is read from external memory over AXI4-Stream.

As this approach offers some challenges, reading the values from a txt-file has proven to be less complicated when simulating the scaling module. To read from a txt-file, the textio-library has to be included in Vivado. The library includes several functions as read, readline, write, writeline, open file and close file. These functions are only for simulation use as they are not synthesisable. They do not require the timing restrictions that come with the AXI communication, but they are helpful for creating and testing the video scaler. The functions of textio can be replaced by AXI and a FIFO for implementation.

4.2 Image-text conversion

The conversion from PNG to txt, was executed using Python. To use the skimage-library which enables the conversion, both the scipy-library and numpy-mkl library are needed. All of the libraries can be downloaded from Gohlke [8]. The following code is the Python code used to convert from image to text.

As the skimage-library use four values, where every forth is 255, which is the maximum value for pixels with 8-bit depth, it was not needed, and therefore not written to the text-file. The three other values are R or red-layer, the G or green-layer and the B or blue-layer.

```
import skimage
import skimage.io

i = skimage.io.imread('input_image.png')
p = 0
h = open ('image2text.txt', 'w')
for pixelVal in i[:,:,:].flat:
                p = p + 1
                if (p < 4):
                        h.write(str(pixelVal) + '\n')
                if (p == 4):
                        p = 0
h.close()
```

4.3 Text-image conversion

After the image has been scaled up or down, another Python program is used to convert it from text to image. Note that three lines are placed at each location in the matrix. It is important to change the numbers inside the imageDimensions-matrix to the intended resolution of the new image.

In this case, it is the image in figure 4.7, where the 480x480 pixel picture has been scaled down to 240x240 pixels. The image that is created already be at least of the same pixel size, as writing to a smaller image will cause errors.

```
import skimage
import skimage.io

inputImage = skimage.io.imread('empty_image.png')
h = open ('text2image.txt', 'r')

imageDimensions = [240,240] #Change to the desired dimensions
for i in range(0, imageDimensions[0]):
        for j in range(0, imageDimensions[1]):
                pixelValue = int (h.readline())
                inputImage [i][j][0] = pixelValue
                pixelValue = int (h.readline())
                inputImage [i][j][1] = pixelValue
                pixelValue = int (h.readline())
                inputImage [i][j][2] = pixelValue

skimage.io.imsave('scaled_image.png', inputImage)
h.close()
```

4.4 Drawn image

Figure 4.4 is one of the four images that have been used in this project. This image was originally 128x64 pixels, and it was scaled down to 64x32 pixels and scaled up to 256x128 pixels. The image was created in the program paint and saved as a PNG-file.

Figure 4.5 shows a monochrome version of the image, where all of the RGB-values have been set to the same value between 0 and 255. All colors will appear "colorless" or monochrome when using the same value on the three different values in the RGB color-space. The image was created when exploring the skimage-library and it gave an illusion that the library used YCbCr instead of RGB, but figure 4.6 shows the three different layers of the image, which is red, green and blue.

Figure 4.4: 128x64 pixel image in original size

Figure 4.5: 128x64 pixel image - Monochrome

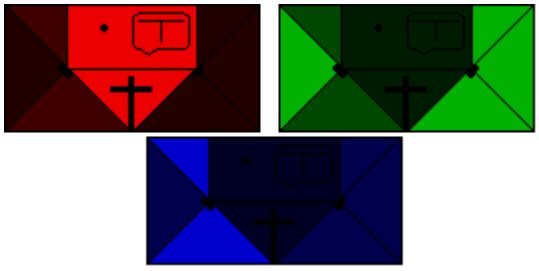

Figure 4.6: 128x64 pixel image - R, G and B layers

4.5 Picture image

The second image used in this thesis is a picture in order to have tested both a drawing and a real picture. The picture in figure 4.7 is larger than the one in figure 4.4, and is easier to use as a benchmark as humans tend to focus on details such as the face to recognize other humans and therefore will be able to tell if the new image has been distorted compared to the original.

Figure 4.7: The original 480x480 pixel picture

The images can easily be exchanged with other images as long as HEIGHT and WIDTH are changed to fit the new image, in both Vivado and in Python.

4.6 Scale2x Algorithm

Scale2x is a pixel art scaling algorithm which was originally developed to improve quality of old video games. The algorithm runs in real-time and it is best suited for enhancing 2D graphics. There are some similarities between the Scale2x-algorithm and an algorithm called EPX, where both will produce the same output, but EPX runs considerably slower (Mazzoleni [9]).

The Scale2x algorithm will do a repeating computation pattern for every pixel of the original image. The pattern starts from a square of 9 pixels and expands the central pixel computing 4 new pixels. Figure 4.6 is an illustration of how the center pixel, E, expands into four pixels, $E0$, $E1$, $E2$ and $E3$.

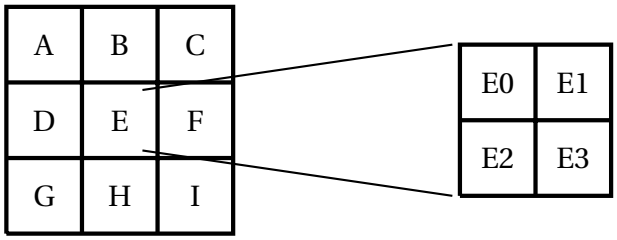

Figure 4.8: Upscaling with Scale2x-Algorithm

The color of each pixel is calculated by the algorithm below (written in C). The algorithm will check if the surrounding pixels are equal, to determinate the value of each of the new pixels.

```
if (B != H && D != F)
{
        E0 = D == B ? D : E;
        E1 = B == F ? F : E;
        E2 = D == H ? D : E;
        E3 = H == F ? F : E;
}
else
{
        E0 = E;
        E1 = E;
        E2 = E;
        E3 = E;
}
```

Besides Scale2x there are two other versions; Scale3x and Scale4x. Scale3x increases the number of pixels by 3x3 (9 pixels) and Scale4x increases the number of pixels by 4x4 (16 pixels). Image 4.11 is the result of using Scale2x and Scale4x on an image from an old video game, Mazzoleni [10]. The algorithms increases the resolution of the image and use the added pixels to make the edges smoother.

The main problem with Scale2x is that it do not care about the values of the pixels which are placed diagonally relative to the currently scaled pixel, which leads to the "wave"-effect that can be seen in figure 4.17.

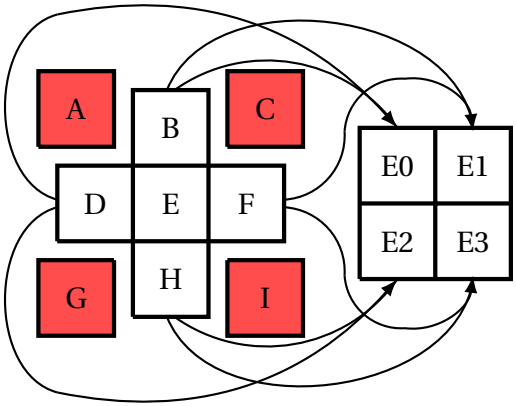

Figure 4.9: Scale2x transition

If the diagonal pixels were included in the calculation of the new pixel, the total number of overlapping values used between neighbouring pixels to decide the values of the new pixel would increase from 2 to 6. On the left in figure 4.6, the current version of scale2x uses both of the gray pixels, *E* and *F* in the upscaling of two neighbouring pixels. On the right, the gray pixels are the pixels that would be included by two neighbouring pixels when calculating the values of the upscaled pixels. Since the number of overlapping pixels have increased, the probability of the new pixels in both neighbours being somewhat equal have increased, and it may remove the "wave"-effect caused by Scale2x. None of these statements have been proven as it has not been tested.

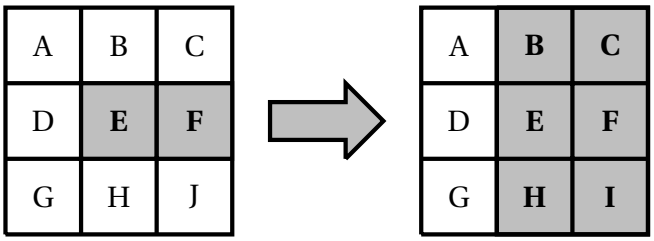

Figure 4.10: Scale2x neighbouring pixels

Figure 4.11: Image of Scale2x and Scale4x, source Mazzoleni [10].

The scale2x algorithm does not always provide satisfactory results as can be seen in image 4.20, where the different scale-algorithms smoothen out edges of objects which are supposed to have sharp edges.

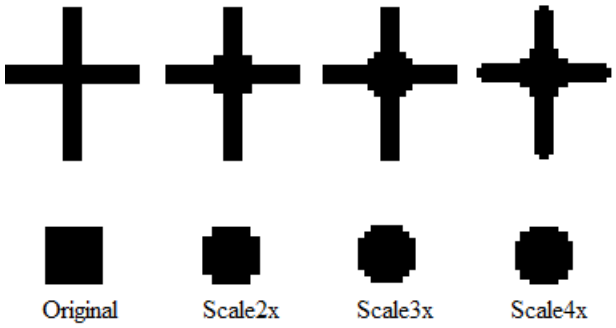

Figure 4.12: Image of Scale2x, Scale3x and Scale4x from Mazzoleni [9].

The process of scaling up is more complex than scaling down in VHDL, especially when implementing the scaler into the AXI4-Stream due to timing the scaler to match AXI4-Stream. Writing four times the amount that is read will require a FIFO in order make sure all data is received and sent in the correct order. The FIFO can either be implemented in the block diagram as a Xilinx IP or implemented directly into the read/write module.
For the simulation, three rows of pixels were used as algorithm uses data from all three in order to decide the color of the new pixels. For the picture 4.7 the number of pixels in one row was 480, but since each pixel had 3 values, the rows had a length of 1440 elements. As figure 4.6 shows, three rows were needed in use the scale2x-algorithm. The algorithm would use a $column \ mod \ 3 \ = \ 0$ calculation in order to read all three layers of RGB before use the pixel in any calculation. In total, scale2x was tested with four variations:

1. Red scaling, where green and blue will not affect the result.
2. Red scaling, where Y-values with less than 10 in value-difference would be treated as equal.
3. Scaling based on all three layers
4. Scaling based on all three layers, allowing a difference of up to 10 as equal

All of the different implementations created some kind of variation when implemented, which can be seen in figure 4.18. When the different variations were tested on a picture, the differences were insignificant and they all made the picture look worse than it did when it was just enlarged. The red-layer scaling was mostly used to test how much of a difference one calculation would make compared to three and the results did not differ much.

Figure 4.13: Reading from memory and storing in arrays for fig 4.7

The order of the rows will rotate based upon which row contains the newest data, where, as in figure 4.6 the bottom row will always contain the newest data, and the top row will contain the oldest data and therefore be exchanged it with new data when the next row will be read.

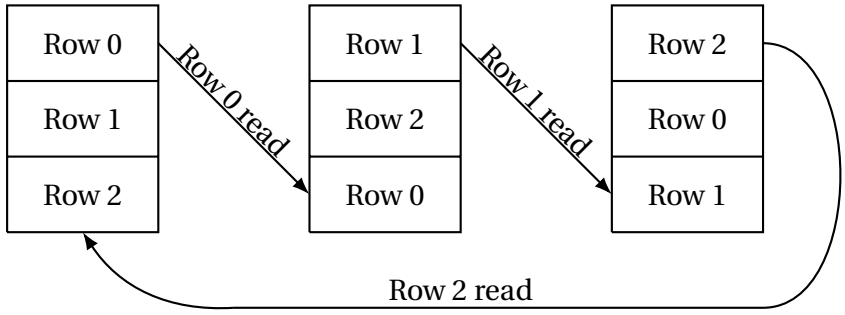

Figure 4.14: Order of which row receives data

When writing the scaled up image to memory, the reading will be suspended to ensure that none of the rows get new data when their current values are being used by the algorithm.

The data that has been scaled up will be stored in six new arrays. The new arrays are R_0, R_1, G_0, G_1, B_0 and B_1, all of which will store up to 2 $WIDTH-1$, which in figure 4.7 is 960 elements. The data have been divided between R, G and B to keep the arrays from containing too much data as simulation in Vivado will yield the following warning:

WARNING: Can't add object to the wave window because it exceeds the display limit of 65536 bits.

The problem does not cause any problem for the implementation of the design, so the different arrays are a workaround for simulation only.

R_0 and R_1 are the red-values of each their row, where R_0 is the top row and R_1 on the bottom row. From figure 4.6, $E0$ and $E1$ will be stored in R_0 while $E2$ and $E3$ will be stored in R_1. G_0, G_1, B_0 and B_1 will likewise store their associated green and blue values respectively.

READ	Row_0(0)	Row_0(1)	Row_0(2)	···	Row_0(1437)	Row_0(1438)	Row_0(1439)
WRITE	R_0(0)	G_0(0)	B_0(0)	···	R_0(959)	G_0(959)	B_0(959)
WRITE	R_1(0)	G_1(0)	B_1(0)	···	R_1(959)	G_1(959)	B_1(959)

READ	Row_1(0)	Row_1(1)	Row_1(2)	···	Row_1(1437)	Row_1(1438)	Row_1(1439)
WRITE	R_0(0)	G_0(0)	B_0(0)	···	R_0(959)	G_0(959)	B_0(959)
WRITE	R_1(0)	G_1(0)	B_1(0)	···	R_1(959)	G_1(959)	B_1(959)

Figure 4.15: Order of reading and writing

Figure 4.6 shows the rate between reading and writing in the program. It will read one row and write two back after it has been through the algorithm. The the pixels that are at the edge of the image, along with their neighboring pixels will copy their data to all four pixels of the new image. That means the first two rows at the top the image and the last two rows at the bottom as well as the first two elements and last two elements of each row, as illustrated in figure 4.6.

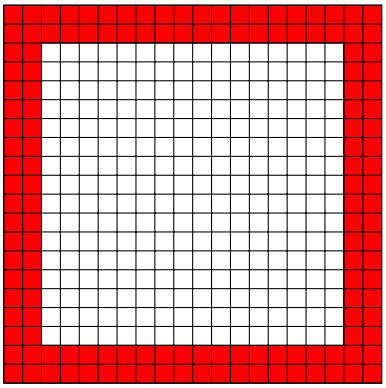

Figure 4.16: Edge pixels

These pixels are expanded without any algorithm for two reasons, which is practicality and timing. The timing is the main reason as any sort of algorithm would require the program to read two rows before writing. The data in these two rows would most likely not be of much significance as most pictures try to center around the main object and the edges are often overlooked.

4.6.1 Results

The results of implementing the Scale2x algorithm has not always been satisfactory and it was tested on four different images.

In picture 4.17 the weakness of Scale2x is clear, as the image appear more pixelated and the straigth lines from picture 4.7 such as on the arm has turned into "waves". The image has also been distorted in such a way that it makes the eyes look like they view in different directions. As previously stated, the different variations of scale2x did not create any significant differences. The image was scaled from 480 x 480 pixels to 960 x 960 pixels.

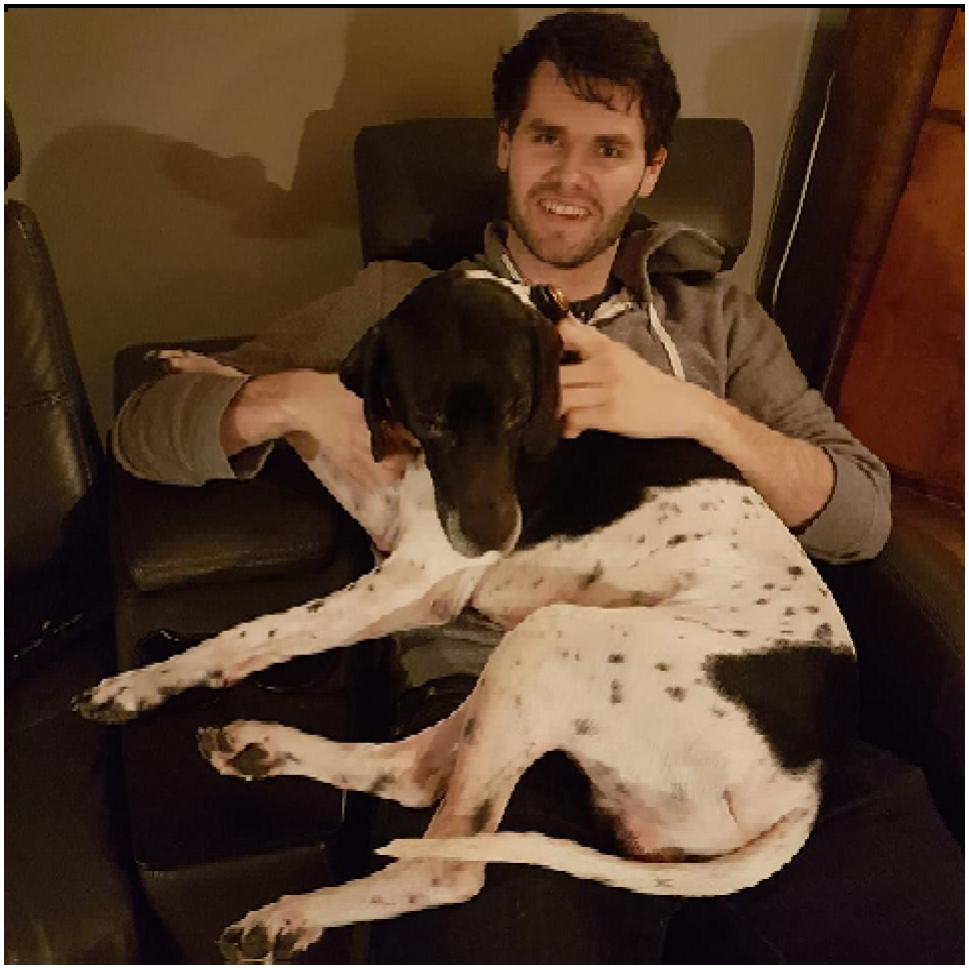

Figure 4.17: Scale2x of picture

Figure 4.18 is just a small snippet of an image to show some details when using Scale2x.

The image was scaled from 640 x 480 pixels to 1280 x 960 pixels and the snippets are around 90 x 90 pixels each.

Figure 4.18: Scale2x with some variations

Image	**Descrition**
Top Left	The original Mario image, resized to the same size as the others
Top Middle	Scale2x based on the red layer
Top Right	Scale2x, based on red layer, accepts difference in R-values less than 10 as equal
Bottom Left	Scale2x, based on red, green and blue
Bottom Right	Scale2x, based on red, green and blue and accepts difference in R, G and B values less than 10 as equal

Table 4.2: Explaination of figure 4.18

The differences in the Mario-images are not easy to spot, but all the scaled images seem less blurry than the original at the top left iamge. The top middle image is the one that base the algorithm on the red layer, which have caused some pixels to stand out, like the two black pixels at the left side of the left eye.
The top right image is the same as the middle, but with some tolerance to the difference in pixel-values.
This improved the quality in regards to the middle as only one pixel stood out, which is the black pixel in the green grass behind Mario.
The image at the bottom left is the same as the top middle, but the algorithm includes calculation of all the layers, Red, Green and Blue, making all the values of RGB equally important. The "wave"-problem that was described in figure 4.17 was not present in any of the scaling attemps for the Mario picture, which indicate that the algorithm is more suitable for low-resolution videogame images.
Figure 4.19 was the figure that was most affected by the variations. The top left is the original image, while the three others are different versions of scale2x. Note that the black line on the right side has shifted in some pixels at the top right, faded on the bottom right, and the line is completely gone on the bottom left.
The images has also had some distortion in the speech bubble and the dot. The top right, which is the luma layer scaler, has also had some pixels sticking out at the cross, the "T" and the dot.

4.7 Hqx Algorithm

Another algorithm that is used in image processing is the hqx, which stands for high quality magnification.

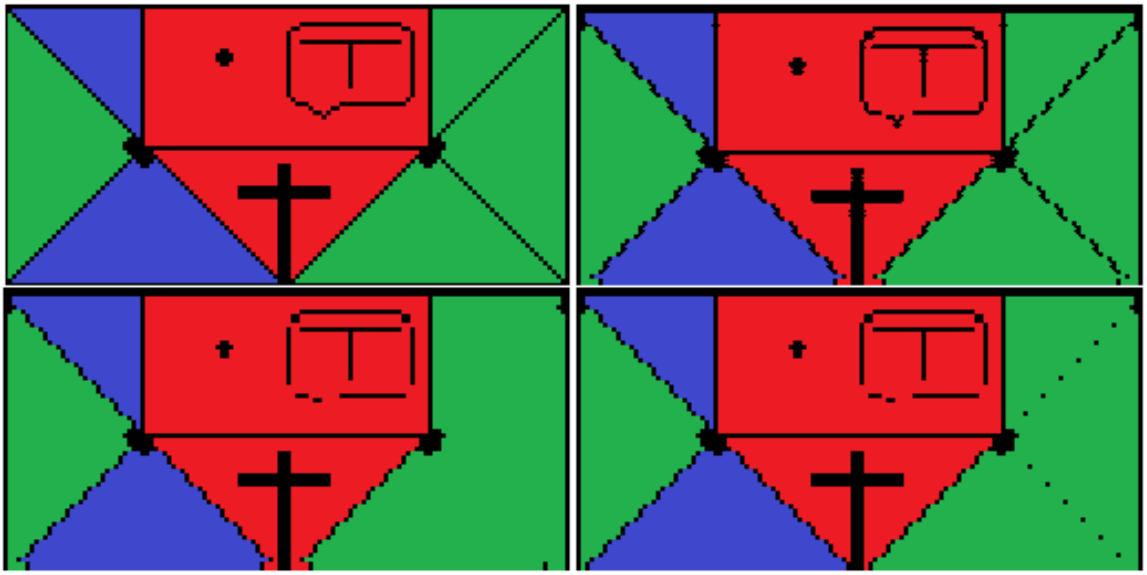

Figure 4.19: Four variations of Scale2x

Hqx is used in several emulators and it has three filters, $hq2x$, $hq3x$ and $hq4x$, which magnify by a factor of 2, 3 and 4 respectively. Just as the Scale2x-algorithm, the hq2x-algorithm will check all of the 8 pixels surrounding it, but it has far more complicated, enabling up to 2^8 different combinations for each pixel. The algorithm detects shapes by checking for pixel of similar colors. When expanding by 2x2, 3x3 or 4x4, the algorithm uses LUTs (lookup tables), which are generated relatively slow, but it makes the render stage fast. Figure 4.20 is a comparison of nearest-neighbor interpolation and hq3x, both of which can be implemented instead of scale2x in the future.

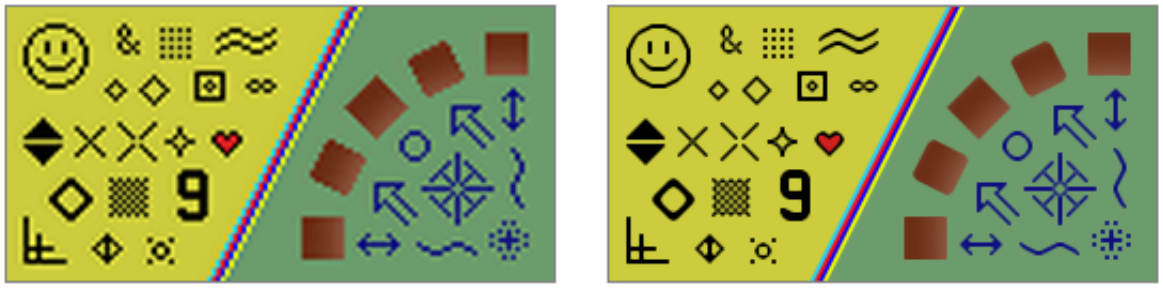

Figure 4.20: Left: 3x enlarged with nearest-neighbor interpolation. Right: 3x enlarged with hq3x

4.8 Downscaling

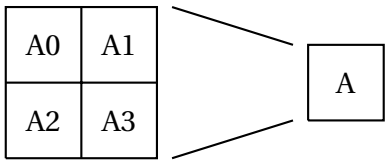

Figure 4.21: Downscaling of 4:1 ratio

The process of scaling down is easier than scaling up as the downscaler will have a lower data flow, which makes it easier to implement on the FPGA. The ratio of data in:out is 4:1 for this project, and a few different implementation have been tested. The program will read two rows of pixels, scale it down to one fourth of the original size and write one row back to memory. As the data of each row is only used once, the memory space which the previous rows occupied in the program can be overwritten by the next two rows of pixels. Figure 4.22 illustrates how the program read from memory and writes back the scaled down data. The equation used for downscaling is:

$$X(A) = \frac{X(A0) + X(A1) + X(A2) + X(A4)}{4} \tag{4.3}$$

READ	row_0(0)	row_0(1)	row_0(2)	···	row_0(381)	row_0(382)	row_0(383)
READ	row_1(0)	row_1(1)	row_1(2)	···	row_1(381)	row_1(382)	row_1(383)
WRITE	R(0)	G(0)	B(0)	···	R(381/6)	G(381/6)	B(381/6)

READ	row_0(0)	row_0(1)	row_0(2)	···	row_0(381)	row_0(382)	row_0(383)
READ	row_1(0)	row_1(1)	row_1(2)	···	row_1(381)	row_1(382)	row_1(383)
WRITE	R(0)	G(0)	B(0)	···	R(381/6)	G(381/6)	B(381/6)

Figure 4.22: Read/Write Ratio for scaling down figure 4.4 for simulation

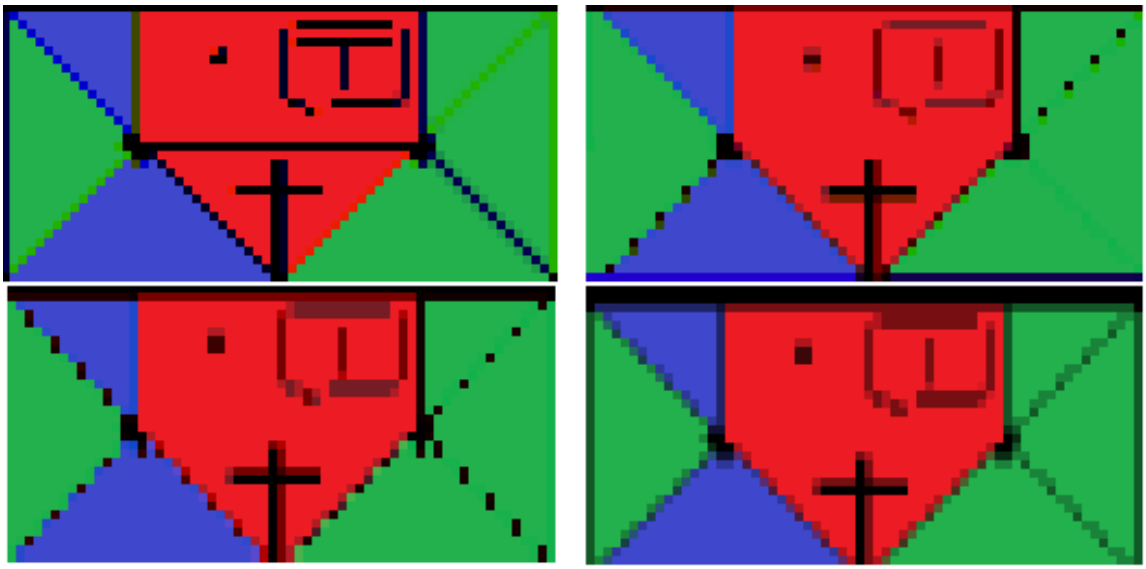

Figure 4.23: Scaled down versions of figure 4.4 scaled down to 64 x 32 pixels

In figure 4.23, four different versions of downscaling is presented. The images have been scaled down from 128x64 pixels to 64x32 pixels.

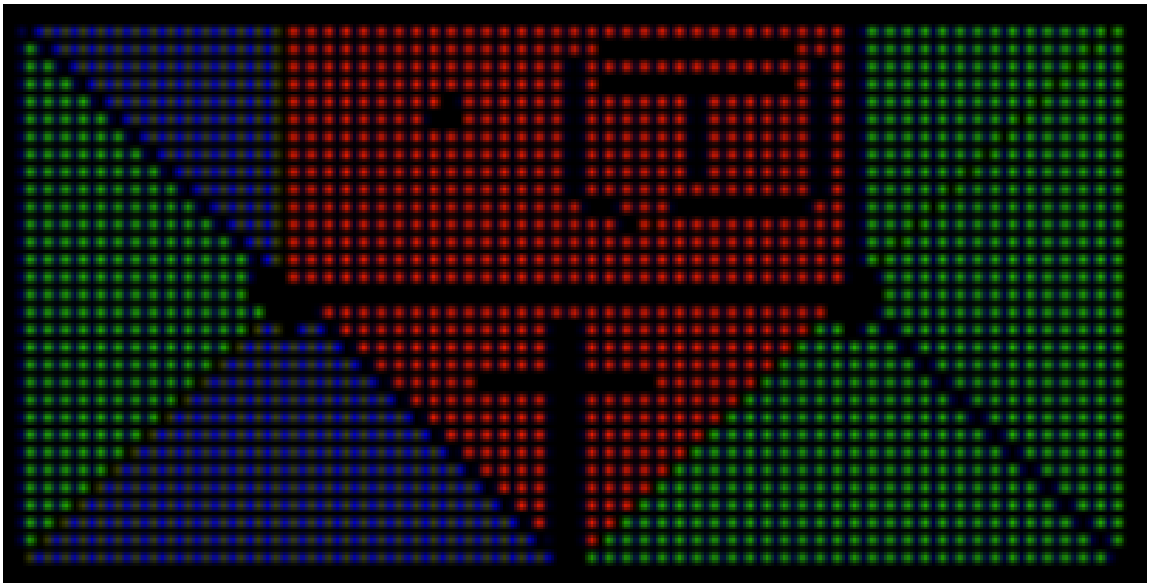

Figure 4.24: Setting the RGB values of 3 out of 4 pixels to 0

The top left image was the first version of the downscaler, where every second row and column were removed, which is why some of the black lines have completely disappeared, while others have not been altered in any way. Figure 4.24 is the same image as the top left of figure 4.23, where the removed pixels have been turned black instead. The lines that are discolored is due to timing error.

The top left was made by adding the R-value of the four pixels, $A0$, $A1$, $A2$ and $A3$ as presented in figure 4.8 and divide it by four to pass the new R-value to A. The G and B values are passed on from the $A0$ pixel, which is why some of the black lines have disappeared. The lower left image is almost identical to the top right, but it bases G and B on the $A2$ pixel, which in this instance gave a result that looked more like the original than the top right. For larger images, this subtle change wont make enough of a difference to be noticed.

The bottom right image is a scaled down version where the RGB-values of A is based on equation 4.3. The X represents R, G or B as all use the same calculation respectively. Some of the noticable differences when scaling with all layers is that the black lines does not disappear, but it gets "watered" down since each pixel is a mixture of all the surrounding pixels.

Figure 4.25: Downscaling by removal

Figure 4.25 and 4.26 are two versions of the downscaled image of Mario. They are relatively equal in terms of how they look, but figure 4.25 has been scaled down by keeping one in four pixels (the $A0$ pixel) and removing the three others completely. Figure 4.26 is scaled down by using equation 4.3 with the R-values of the four pixels, $A0$, $A1$, $A2$ and $A3$, and copy the G and B values from $A0$. The last version is figure 4.27 which yield the best result in regards to the quality. The image is downscaled by using the 4.3 on R, G and B.

Figure 4.26: R-based downscaling

To get a closer look on the details of the image, one enlarged screencap from each of the three figures can be seen in figure 4.28 where the right image, which is from the RGB downscaled version, has more details on the red outline of the back.

Figure 4.27: RGB-based downscaling

Figure 4.28: Left: Removal, Middle: R-scale, Right: RGB-scale

The three images 4.29, 4.30 and 4.31 are as the previous images downscaled by removing 3 out of 4 pixels, downscaled by R-value and keeping G and B from $A0$, and downscaled by RGB-values, respectively. All of the images was scaled from 480x480 pixels to 240x240 pixels.

Figure 4.29: Reduction by removal

Image 4.29, or the reduction-version have two major flaw compared to the RGB-downscaled version, which is the pixelated look of the image, giving the impression of lower resolution than it actually have. The second flaw is the visible transition between vertical rows, which is caused by removing every second row. When implementing the red-layer downscaler, the quality improves, as seen in figure 4.30. The vertical transitions are mostly invisible, but the pixelated look is still present in several places such as the left arm.

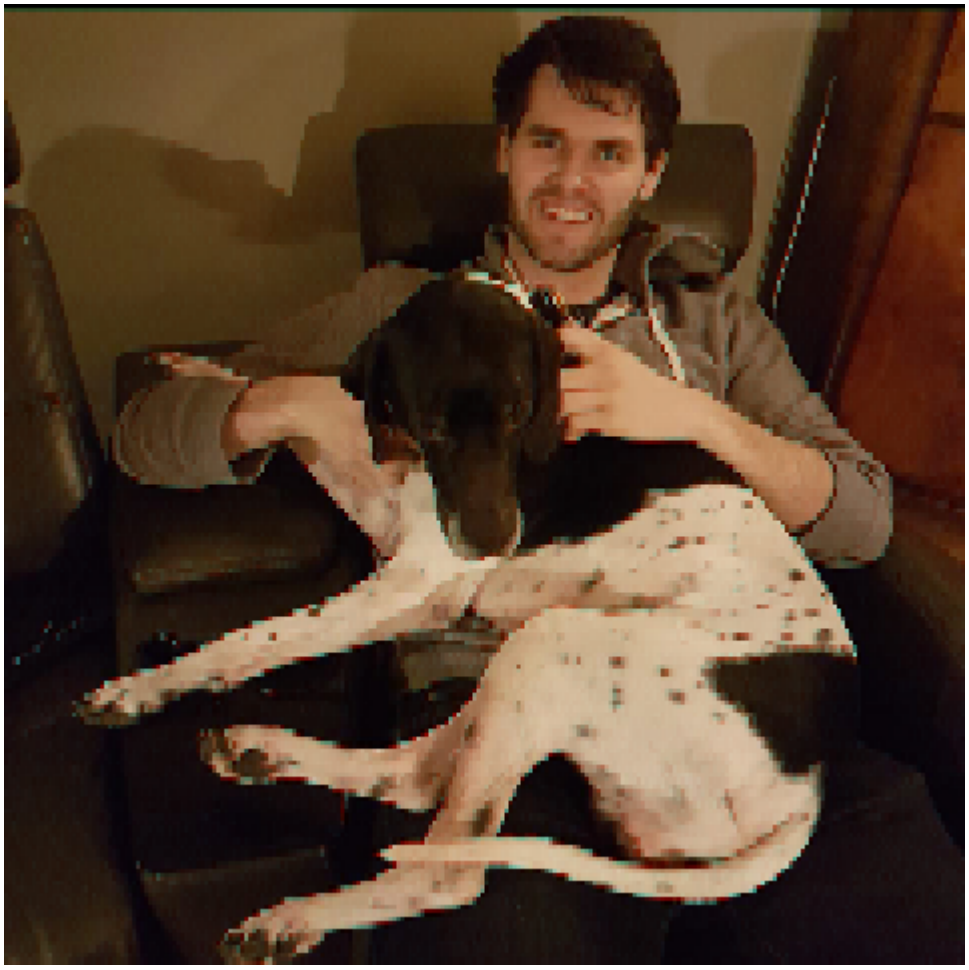

Figure 4.30: Reduction by R-downscaling

The RGB-downscaler, image 4.31 provided the best result. The image still have some pixelated areas, but the quality has improved in comparison to the two other implementations.

Figure 4.31: Reduction by RGB-downscaling

Figure 4.32 is a close up snippet of the face in the three scaled down images. The left is the reduction-version, the middle is the R-downscaled version and the right is the RGB-downscaled version.

The last image, 4.33, show in three different sizes, where the right image is the original 512 x 512 pixels, the middle is the scaled down 256 x 256 pixel image and the left is the 1024 x 1024 pixel image. The images are displayed to show how the images actually look next to each other.

Figure 4.32: Close-up image of the head in downscaled picture

Figure 4.33: Lena

The upscaled image showed good results in terms of quality compared to some of the other images.

Chapter 5

Discussion and Results

The main goal of this thesis was to implement a video scaler module into an AXI-Stream video system which is suited to run on a FPGA. The final result is divided into several parts, as the different parts of the system does not yet fit together. The communication-system, the video system and the two scaling modules are therefore kept separate as it will requires more work to fit it all together. The downscaler was implemented into a premade AXI-DMA system for testing on the FPGA since the system did not work properly.

5.1 Video Scaler

The video scaler has been the part with the most freedom of choice in this project. When choosing which algorithm to use, it was important to have something that would be fast enough for the timing constraints and provide good results in terms of visual quality and utilization.

5.1.1 Upscaler

The upscaling algorithm, Scale2x, is fast and it was intended for old video games with low resolution, but the algorithm provided good results on pictures as well. As the upscaler has not been implemented into an AXI4-Stream, the timing constraints and utilization can not be calculated, but the utilization of the downscaler in the DMA Interrupt example in figure 5.9 show only a small percent is used compared to what is available on the Zedboard, which indicates that the utilization will not be the limiting factor.

For the timing constraints, the performance is likely to be critical, since the downscaler needed to decrease the frequency of the clock in order meet the timing constraints as seen i figure 5.10 and figure 5.11. It can therefore be expected that the upscaler will need to run at a lower frequency than the downscaler as it calculates the values of 4 pixels each cycle, while the downscaler calculates one pixel for every other cycle. In order to increase the performance of the upscaler, a second clock with twice the frequency of the first should be added. The purpose of the second clock is to enable faster output of data as it is likely to be the bottleneck of the upscaler. Other techniques which should be implemented is pipelining, so that the clock can stay at a higher frequency while the upscaler divides the work between multiple cycles.

Figure 5.1 is a snippet of the simulation of the upscaler where row_0 is filling the last slot (1535) at 740,831 ns, resulting in col resetting from 1535 to 0, setting $suspend$ to 1, and updating all the output-rows, R_0, R_1, G_0, G_1, B_0 and B_1 and sending all of the updated data. This is only possible in the simulation as AXI4-Stream is not able to send 6144 values in one cycle. When the outputs are sent, $suspend$ is reset, increasing row to 241 and the simulation start reading data to row_1. See appendix B for the code for the upscaler.

The following equation is used to calculate the theoretical maximum frequency based on current period time and worst negative slack.

$$F_{max}\ (Mhz) = \frac{1}{\frac{Clk\_period \times 10^{-9} - WNS \times 10^{-9}}{1\ 000\ 000}} \tag{5.1}$$

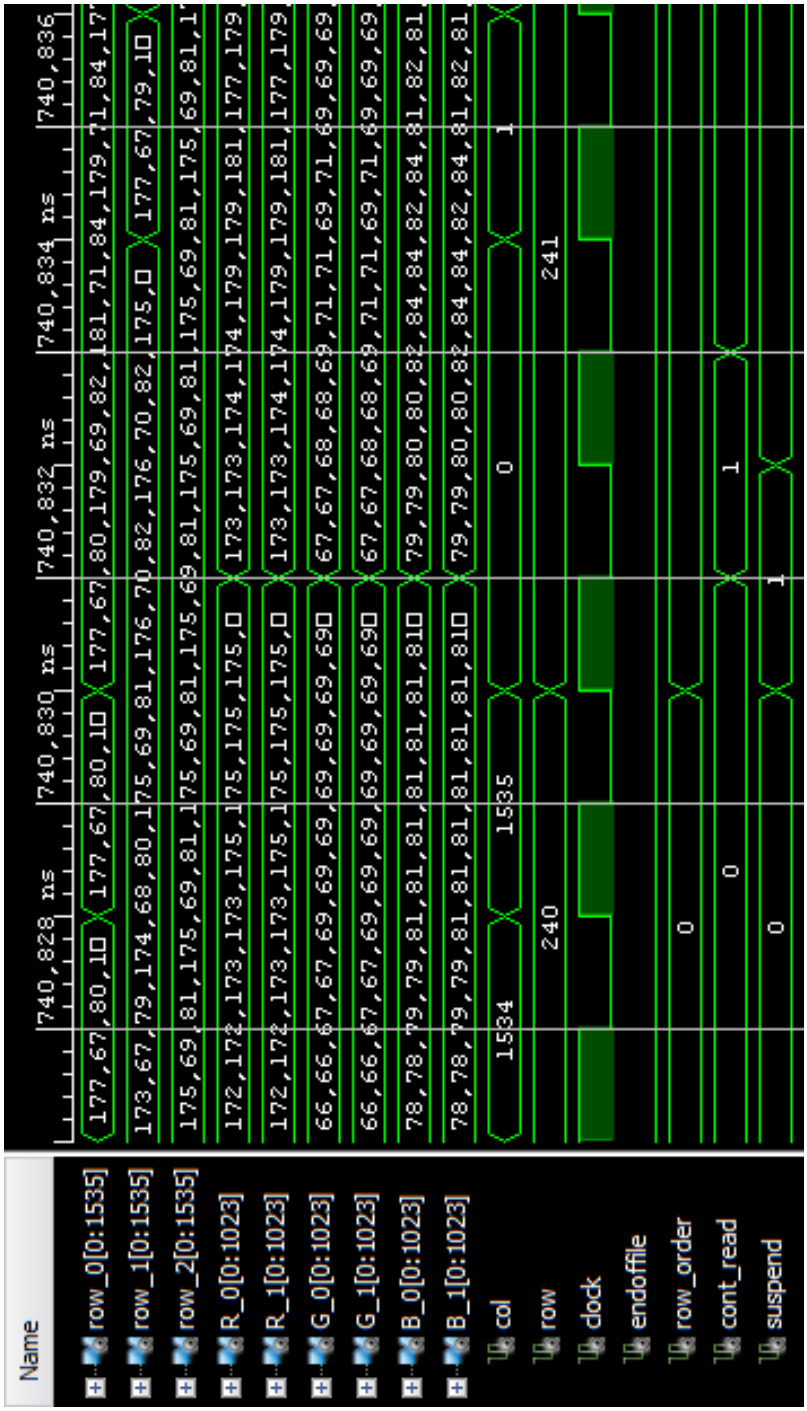

Figure 5.1: Simulation of upscaler on 512 x 512 image

5.1.2 Downscaler

The downscaler was both simulated and implemented, providing visual results, which can be found in chapter 4, as well as utilization and timing report. The code for the implemented scaler can be found in appendix A.1 and the code for the simulated scaler can be found in appendix A.2. The AXI DMA system uses a 100 Mhz clock, and therefore the goal was to create a scaler that was able operate at the same frequency.

The first implementation of the downscaler focused on functionality over efficiency which forced the period time up to 50 ns. The results can be seen in figure 5.2, where the only number standing out is the number of input/outputs, since it is 23%. The number does not matter, since most of the I/O-ports are just needed for simulation. The numbers indicate how much of a Zedboard FPGA the design would occupy.

The utilization has the same amount of flip-flops as both the utilization for 30 ns and 20 ns, which is due to both of the implementations using the same calculations.

Resource	Utilization	Available	Utilization %
FF	99	106400	0.09
LUT	1224	53200	2.30
Memory LUT	722	17400	4.15
I/O	46	200	23.00
BUFG	1	32	3.12

Figure 5.2: Utilization of downscaler with 50 ns period

Since timing constraints were not considered in the first implementation, and the maximum frequency of the implementation was 21,33 Mhz calculated by using equation 5.1. This is equal to a period of rougly 46,89 ns, meaning that a faster period would not meet timing constraints.

Figure 5.3 shows the timing constraints for the implemented downscaler at a 50 ns period. The second implementation, which met timing constraints set to 30 ns, was a

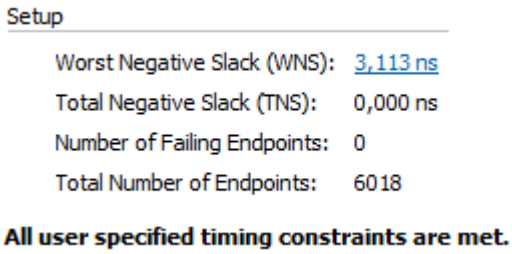

Figure 5.3: timing of of downscaler with 50 ns period

AXI-Stream version of the simulation downscaler from appendix A.2 with some added improvements. The utilization of this implementation can be found in figure 5.4. The main difference between this implementation and the previous, was the increased number of LUTs and the reduced number of I/Os. The timing of the second imple-

Resource	Utilization	Available	Utilization %
FF	99	106400	0.09
LUT	1251	53200	2.35
Memory LUT	722	17400	4.15
I/O	44	200	22.00
BUFG	1	32	3.12

Figure 5.4: Utilization of downscaler with 30 ns period

mentation was improved compared to the first, as it reduced the timing constraints to 30 ns, while having over 7 ns of slack as seen in figure 5.5. It would have been safe to reduce the period to 25 ns since the minimum slack was over 23%, meaning that even the slowest of operation would spend 23% of a cycle waiting. The maximum frequency was 44,25 Mhz and the fastest clock period would have been 22,60 ns.

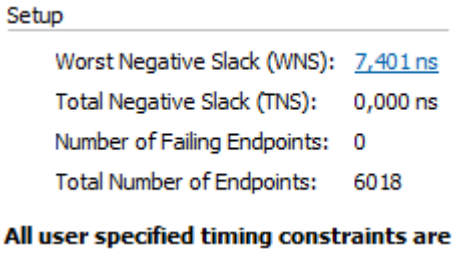

Figure 5.5: timing of of downscaler with 30 ns period

The final implementation of the downscaler can be found in appendix A.1. For this design, the read/write-ratio was altered by putting both functions in the same process. The bottleneck of this implementation is the read-function as the scaling and writing have to wait for the read-function in order to continue. To improve this further, a separate clock should be added to the input AXI-Stream at double the frequency of the rest of the system, and some pipelining to make the downscaling into a two-cycle process rather than one cycle at half the frequency.

Figure 5.6 is the final utilization of the downscaler and the results show low usage which is good.

Resource	Utilization	Available	Utilization %
FF	99	106400	0.09
LUT	1236	53200	2.32
Memory LUT	768	17400	4.41
I/O	44	200	22.00
BUFG	1	32	3.12

Figure 5.6: Utilization of downscaler with 20 ns period

Figure 5.7 is the timing constraints of the final implementation. The implementation used a 20 ns clock period, which was the best achieved results for the downscaler. The maximum frequency is 57,81 Mhz, which gives a minimum period time of 17,30 ns.

For an image with a resolution of 512 x 512 pixels, the theoretically achievable frames per second would be around 63 frames per second with a 50 Mhz frequency.

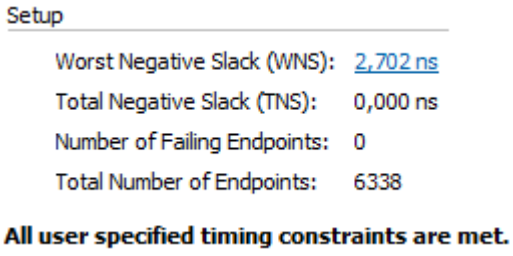

Figure 5.7: timing of of downscaler with 20 ns period

Figure 5.8 is the simulation of the downscaler. The image on top shows when the first row of pixels has been read by the module. All arrays are empty except for row_0, until the time is 3,073 ns, which is when row_1 is filled up. The row starts to fill one cycle after row is increased from 0 to 1. There is no writing at this stage since the simulation will fill row_1 before it writes. The bottom image shows when row_1 is filled when the time is 6,144 ns. The row number increases from 1 to 2 at this point, and the three output-arrays, R, G and B are filled and sent. As stated previously, sending data like this would not be possible with AXI-Stream.

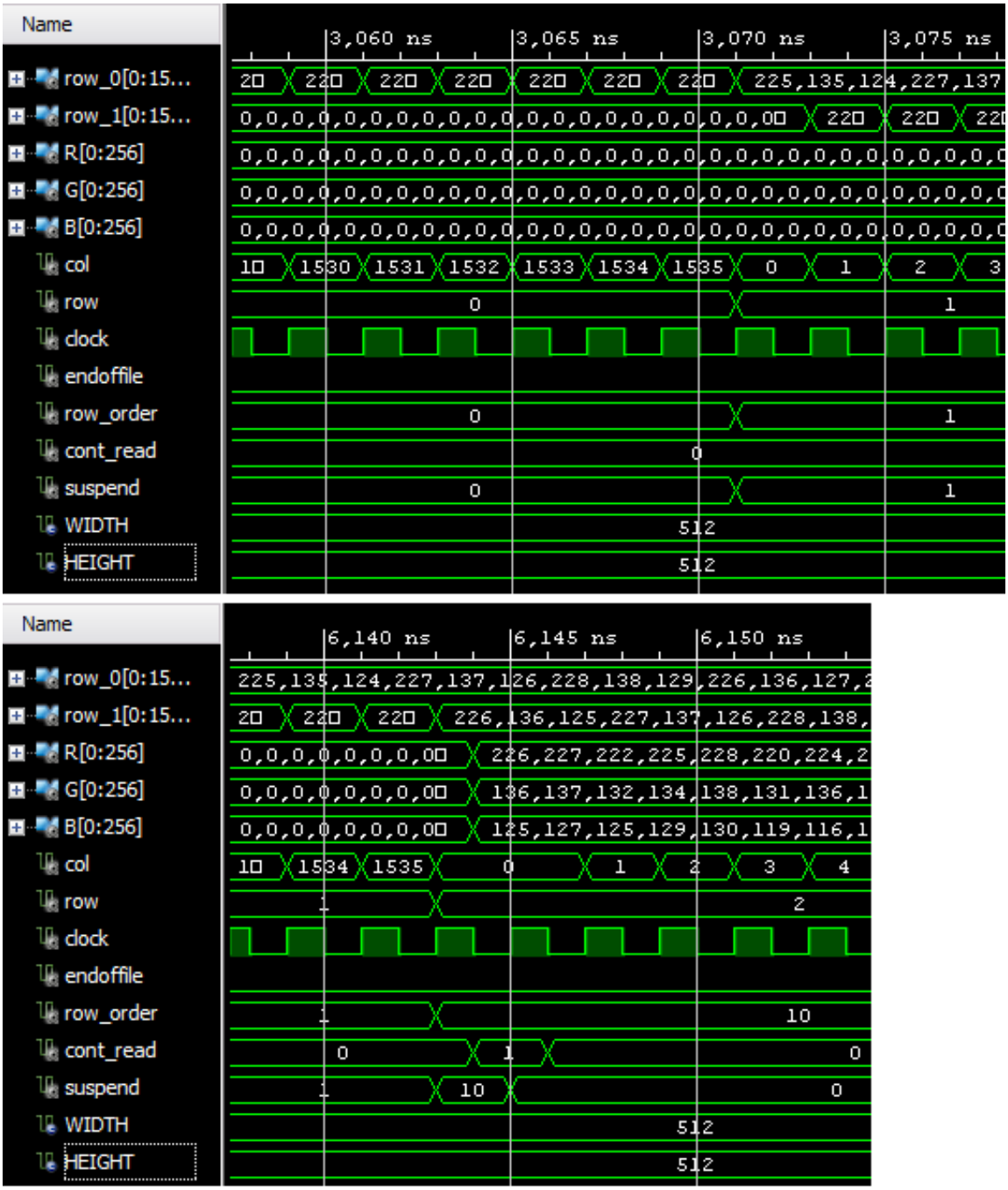

Figure 5.8: Simulation of downscaler

5.1.3 Downscaling DMA System

The downscaler was tested in the DMA Interrupt example, which is referred to as the AXI DMA system, to see how it would fit in an AXI-system. Figure 5.9 shows the number of flip-flops, look-up tables and block RAMs that have been used when the downscaling module have been implemented into the AXI4-Stream DMA example from Xilinx-DMA-ex [19]. The *Available*-column shows the maximum number of the different resources on the Zedboard, while the left column, *Utilization*, shows how many percent of the total resources was used.

Resource	Utilization	Available	Utilization %
FF	4793	106400	4.50
LUT	5765	53200	10.84
Memory LUT	1377	17400	7.91
BRAM	10	140	7.14
BUFG	1	32	3.12

Figure 5.9: Utilization for the downscaler implemented into the DMA interrupt example

Figure 5.10 show two of the timing reports for the design, which was not met with a clock running each cycle at 10 ns or frequency of 100 Mhz. Since the number of failing endpoints in both are fairly small compared to the total number of endpoints it indicates that there may be one part of the design which slower than the rest. The timing report in the right side has some improvements compared to the left, but none of them are efficient enough to run at 100 Mhz.

To calculate what the maximum frequency of the design was, equation 5.1 was used and the result was 64,89 Mhz for the left report and 70,69 Mhz, which would make the fastest possible clock period just over 15,4 ns for the left report and 14,1 ns for the right report.

To adjust for unforeseen slack that may occur at a later point due to hardware delay which may not have been taken into account, it would be safe to increase the clock period to 20 ns which means a frequency of 50 Mhz.

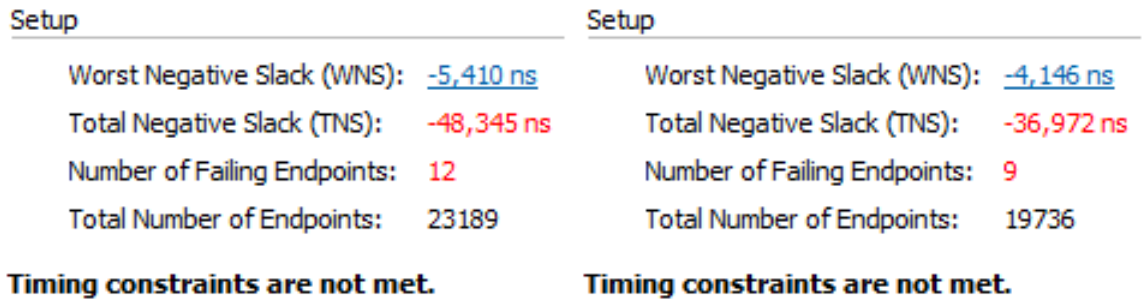

Figure 5.10: Timing constraints for 10 ns clock period

To fix the timing constraints of the design, the Zynq7 processing system must be configured as it is the source of the clock in the design. The frequency was changed from 100 Mhz to 50 Mhz, which changed the clock period from 10 ns to 20 ns, and the results of the change can be seen in figure 5.11, where all timing constraint are met.

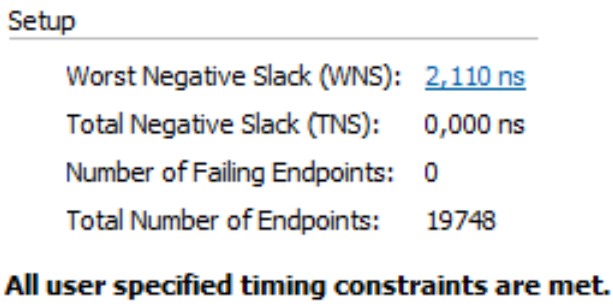

Figure 5.11: Timing constraints for 20 ns clock period

The downscaler that was implemented into the DMA example was more efficient than the one used for simulation, due to an error that could only be bypassed by using Linux according to Xilinx Forum, [18]. This error only occured when using the textio-library, which is not needed in the implementation as it will not read from files files, but rather get access through SDK. The visual results of both the upscaler and the downscaler would not have been any different if the error had not existed, just their efficiency when simulating.

5.2 Design

The design consists mostly of IP cores provided by Xilinx, where most of them can be configured to fit the needs of the system. As for the AXI-communication, the cores can be connected by dragging their pins together. This simplifies the making of the system, but a lot of knowledge is needed in order to make it work as intended. All of the cores are essentially black boxes, but Xilinx provides product guides for all their IPs along with practical examples on how to use some of them.

The video system and the control system did not communicate the way they were supposed to, which was partially due to restricted testing as running synthesis is time consuming and in worst case scenario would run for up to 45 minutes. The block design system was not simulated as it is seen as a black box, leaving no information of the internal signals.

5.3 Environment

The video system was made in Vivado 2015.1 which is the design platform for the AXI DMA example, where the downscaler was implemented. One of the main concerns when using a certain version of Vivado is that some IP cores may have a somewhat different use or they may be replaced by new IPs. Xilinx are in the process of removing XMD, which is essential to build the AXI DMA example, but it has not been removed in Vivado 2016.4 or any earlier versions, at least.

It should be noted that a powerful computer would be preferred when working with Vivado as running synthesis would often take up to 30 minutes to complete. To program the FPGA, Xilinx SDK was used along with PuTTY, which is a serial console used to read the data that is sent out from the FPGAs UART-port.
For the simulation, two different Python programs was used since they offered a fast method to convert the images to text-based files, which in turn worked great with the textio-library of Vivado. The second program reversed the process by converting from text to PNG.

5.4 Future Work

The future work of this project would be to put it all together so that it would fit into the Post-Doctoral project that it is intended to be a part of. The upscaler should be implemented into the video system, as it currently only works in simulation. Checking the differences in the downscaler used for simulation and implementation should provide some guidance on how to do this. The scalers should also be improved in terms of efficiency, as the current versions do have room for improvements. Inserting an additional clock for reading into the downscaler and writing from the upscaler at double frequency as well as pipelining the scalers might be good techniques to achieve faster performance and make the system able to run at a clock period of 10 ns.

Chapter 6

Conclusion

This thesis is a presentation of the different aspects of making a video scaler suited for FPGA implementation. The system uses Xilinx Viviado and Xilinx SDK to create, implement and run the video system on a ZedBoard Zynq-7000 ARM/FPGA SoC Development Board.

Both an upscaler and a downscaler was made, both of which scale by a factor of four. The upscaler uses a Scale2x-algorithm, with some variations, which provided some good results, but it was not implemented on the FPGA due to limited time. The downscaler uses merging to downscale the image and it was implemented on the FPGA by rewriting some of the simulation-functions to fit AXI4-Stream instead. The scalers are easy to reconfigure for other resolutions, and the downscaler can in theory achieve up 63 frame per second for an image of 512 x 512 resolution, though it has not been proven in this thesis. The downscaler is currently fast enough to run at a 20 ns clock period, which equals a frequency of 50 Mhz and as the utilization reports showed, the scaler use a very small amount of what is available, which is great.

The results of the thesis was satisfactory in some aspects, such as the video scaler, but it will need more work to put together the video scaler, the video module and the control module to a complete system. The system uses AXI4-Stream and AXI4-Lite as communication and dataflow between the modules, and the downscaler was proven to work with these protocols.

Bibliography

[1] AXI-FIFO-Xilinx. Axi-stream-fifo,
https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s
/v4_1/pg080-axi-fifo-mm-s.pdf.

[2] AXI-Guide-Xilinx. Axi-guide,
https://www.xilinx.com/support/documentation/ip_documentation
/ug761_axi_reference_guide.pdf.

[3] AXI-Interconnect-Xilinx. Axi-interconnect,
https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect
/v2_1/pg059-axi-interconnect.pdf.

[4] AXI2VIDEO-Xilinx. Axi2video,
https://www.xilinx.com/support/documentation/ip_documentation/v_axi4s_vid_out
/v3_0/pg044_v_axis_vid_out.pdf.

[5] Bovik, A. Handbook of image and video processing
http://iitlab.bit.edu.cn/mcislab/w̃uxinxiao/ppt/handbook%20of%20image%20and%
20video%20processing.pdf.

[6] DMA-Xilinx. Dma, https://www.xilinx.com/support/documentation/

ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[7] FPGA-Developer. Dma in axi-system, http://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html.

[8] Gohlke, C. Python libs, http://www.lfd.uci.edu/~gohlke/pythonlibs/.

[9] Mazzoleni, A. Scale2x, http://www.scale2x.it/.

[10] Mazzoleni, A. Scale2x, http://www.scale2x.it/download.

[11] resampler Xilinx, C. chroma-resampler, https://www.xilinx.com/support/documentation/ip_documentation/v_cresample/v4_0/pg012_v_cresample.pdf.

[12] RGB2YCBCR-Xilinx. Video-on-screen-display, https://www.xilinx.com/support/documentation/ip_documentation/v_rgb2ycrcb/v7_1/pg013_v_rgb2ycrcb.pdf.

[13] VDMA-Xilinx. Vdma, https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf.

[14] VOSD-Xilinx. Video-on-screen-display, https://www.xilinx.com/support/documentation/ip_documentation/v_osd/v6_0/pg010_v_osd.pdf.

[15] VTC-Xilinx. Video-timing-controller, https://www.xilinx.com/support/documentation/ip_documentation/v_tc/v6_0/pg016_v_tc.pdf.

[16] VTPG-Xilinx. Video-test-pattern-generator,

https://www.xilinx.com/support/documentation/ip_documentation/v_tpg/v7_0/pg103-v-tpg.pdf.

[17] Xilinx. Xapp742, axi vdma reference design, https://www.xilinx.com/support/documentation/application_notes/xapp742-axi-vdma-reference-design.pdf.

[18] Xilinx, F. Dma_ex_interrupt, https://forums.xilinx.com/t5/simulation-and-verification/vivado-2014-2-windows-7-xelab-exe-appcrash-problem/td-p/562741.

[19] Xilinx-DMA-ex. Dma_ex_interrupt, https://www.xilinx.com/support/answers/57562.html.

Appendix A

Downscaler

A.1 Implemented Downscaler

```
----------------------------------------------------------------------
-- Company: NTNU
-- Engineer: Tom Erik Tysse
-- Design Name: Downscaler
-- Module Name: Scaler
-- Additional Comments: Implemented downscaling module
-- Remember to change HEIGHT and WIDTH to fit the image
----------------------------------------------------------------------

--include this library for file handling in VHDL.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std;
USE ieee.numeric_std.ALL;
--library std;
--use std.textio.all;


entity scaler is
```

```
  Port (
        s_axis_tdata    : in std_logic_vector(31 downto 0);
        aclk            : in std_logic;
        aresetn         : in std_logic;
        s_axis_tvalid   : in std_logic;
        s_axis_tready   : out std_logic := '0';
        m_axis_tdata    : out std_logic_vector(31 downto 0)
                        := (others => '0');
        m_axis_tvalid   : out std_logic := '0';
        m_axis_tready   : in std_logic;
        m_axis_tlast    : out std_logic
);
end scaler;

architecture Behavioral of scaler is
-- Change WIDTH and HEIGHT to fit image
constant WIDTH      : integer := 512;
constant HEIGHT     : integer := 512;

subtype pixel is integer range 0 to 255;
type mem_type is array (integer range 0 to ((3*WIDTH)-1)) of pixel;

signal row_0        : mem_type;
signal row_1        : mem_type;
signal end_of_file  : std_logic := '0';
signal col          : integer := 0;
signal row          : integer := 0;
signal en           : integer := 0;

begin
endfile : process (aclk)
begin
if (aclk = '1' and aclk'event) then
    if (row = HEIGHT) then
        end_of_file <= '1';
    end if;
end if;
end process;
```

```
read_write : process (aclk)
begin
if (aclk = '1' and aclk'event) then
    if (end_of_file = '0') then
        if (s_axis_tvalid = '1') then
            s_axis_tready <= '1';
            en <= 1;
            if (row mod 2 = 0) then
                row_0(col) <= to_integer(unsigned(s_axis_tdata));
            else
                row_1(col) <= to_integer(unsigned(s_axis_tdata));
            end if;
        else
            en <= 0;
        end if;
        if(row mod 2 = 1) then
            if (col mod 6 > 2) then
              if (col mod 6 = 3) then
                m_axis_tvalid <= '1';
                 m_axis_tdata <= std_logic_vector
                 (to_unsigned((row_0(col-3) + row_0(col)
                 + row_1(col-3)  + row_1(col)) / 4,32));
              elsif (col mod 6 = 4) then
                 m_axis_tdata <= std_logic_vector
                 (to_unsigned((row_0(col-3) + row_0(col)
                 + row_1(col-3) + row_1(col)) / 4,32));
              elsif (col mod 6 = 5) then
                 m_axis_tdata <= std_logic_vector
                 (to_unsigned((row_0(col-3) + row_0(col)
                 + row_1(col-3) + row_1(col)) / 4,32));
                 m_axis_tvalid <= '0';
              end if;
            end if;
        else
            en <= 1;
        end if;
            if (en = 1) then
```

```
                col <= col + 1;
                if (col = 3*WIDTH-1) then
                    col <= 0;
                    row <= row + 1;
                end if;
            end if;
    else
        null;
    end if;
end if;
end process;
end Behavioral;
```

A.2 Simulated Downscaler

```
----------------------------------------------------------------------
-- Company: NTNU
-- Engineer: Tom Erik Tysse
-- Design Name: Downscale_module
-- Additional Comments: Downscaler used for simulation
-- Change HEIGHT, WIDTH and file-path to fit image
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library std;
use std.textio.all;

entity downscale_module is
--  Port ( );
end downscale_module;

architecture Behavioral of downscale_module is
constant WIDTH  : integer := 512;
constant HEIGHT : integer := 512;
subtype pixel     is integer range 0 to 255;
type mem_type     is array (integer range 0 to 3*WIDTH-1) of pixel;
type out_mem_type is array (integer range 0 to WIDTH/2) of pixel;
signal row_0    : mem_type;
signal row_1    : mem_type;
signal R        : out_mem_type;
signal G        : out_mem_type;
signal B        : out_mem_type;
signal col      : integer := 0;
signal row      : integer := 0;
signal clock    : bit := '0';
signal endoffile: bit := '0';
signal row_order: integer := 0;
signal cont_read: integer := 0;
signal suspend  : integer := 0;
begin
```

```
clock <= not (clock) after 1 ns;

read_proc : process
    file      in_file    : text is in
              "C:\Users\Tom\Desktop\in_image.txt";
    variable  in_line    : line;
    variable  data_read    : integer;
begin
wait until clock = '1' and clock'event;
if (not endfile(in_file)) then
if (suspend = 0) then
        readline(in_file, in_line);
        read(in_line, data_read);
        if (row_order = 0) then
            row_0(col) <= data_read;
        elsif (row_order = 1) then
            row_1(col) <= data_read;
        end if;
        col <= col + 1;
        if (col = 3*WIDTH-1) then
            col <= 0;
            row <= row + 1;
            suspend <= 1;
            row_order <= row_order + 1;
            if (row_order = 2) then
                row_order <= 0;
            end if;
        end if;
end if;
else
    endoffile <= '1';
end if;
if (cont_read = 1) then
  suspend <= 0;
end if;
end process read_proc;

write_proc : process
```

```
    file      out_file  : text is out
              "C:\Users\Tom\Desktop\out_image.txt";
    variable  out_line  : line;
begin
wait until clock = '0' and clock'event;
if(endoffile='0') then
    if (suspend = 1) then
        if (row mod 2 = 1) then
        for j in 0 to (3*WIDTH-1) loop
            if (j mod 6 = 0) then
                  R(j/6) <= (row_0(j)+ row_1(j)
                         + row_0(j+3) + row_1(j+3)) / 4;
                  G(j/6) <= (row_0(j+1)+ row_1(j+1)
                         + row_0(j+4) + row_1(j+4)) / 4;
                  B(j/6) <= (row_0(j+2)+ row_1(j+2)
                         + row_0(j+5) + row_1(j+5)) / 4;
            end if;
        end loop;
        for j in 0 to ((WIDTH/2) - 1) loop
            write(out_line,R(j));
            writeline(out_file,out_line);
            write(out_line,G(j));
            writeline(out_file,out_line);
            write(out_line,B(j));
            writeline(out_file,out_line);
        end loop;
        end if;
        cont_read <= 1;
    else
        cont_read <= 0;
    end if;
else
null;
end if;
end process write_proc;
end Behavioral;
```

Appendix B

Simulated Upscaler

```
----------------------------------------------------------------------
-- Company: NTNU
-- Engineer: Tom Erik Tysse
-- Design Name: Upscaler_module
-- Additional Comments: Upscaler for simulation
-- Change HEIGHT, WIDTH and file-path to fit image
----------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library std;
use std.textio.all;

entity upscale_module is
--  Port ( );
end upscale_module;

architecture Behavioral of upscale_module is
-- Change to fit the image before scaling
constant WIDTH    : integer := 512;
constant HEIGHT   : integer := 512;
subtype pixel     is integer range 0 to 255;
type mem_type     is array (integer range 0 to 3*WIDTH-1) of pixel;
type out_mem_type is array (integer range 0 to 2*WIDTH-1) of pixel;
```

```
signal row_0      : mem_type;
signal row_1      : mem_type;
signal row_2      : mem_type;
signal R_0        : out_mem_type;
signal R_1        : out_mem_type;
signal G_0        : out_mem_type;
signal G_1        : out_mem_type;
signal B_0        : out_mem_type;
signal B_1        : out_mem_type;
signal col        : integer := 0;
signal row        : integer := 0;
signal clock      : bit := '0';
signal endoffile  : bit := '0';
signal row_order  : integer := 0;
signal cont_read  : integer := 0;
signal suspend    : integer := 0;

begin

clock <= not (clock) after 1 ns;

read_proc : process
    -- Insert file path
    file      in_file    : text is in
              "C:\Users\Tom\Desktop\in_image.txt";
    variable  in_line    : line;
    variable  data_read  : integer;
begin
wait until clock = '1' and clock'event;
if (not endfile(in_file)) then
    if (suspend = 0) then
        readline(in_file, in_line);
        read(in_line, data_read);
        if (row_order = 0) then
            row_0(col) <= data_read;
        elsif (row_order = 1) then
            row_1(col) <= data_read;
        elsif(row_order = 2) then
```

```
            row_2 (col) <= data_read;
        end if;
        col <= col + 1;
        if (col = 3*WIDTH-1) then
            col <= 0;
            row <= row + 1;
            suspend <= 1;
            row_order <= row_order + 1;
            if (row_order = 3) then
                row_order <= 0;
            end if;
        end if;
    end if;
else
    endoffile <= '1';
end if;
if (cont_read = 1) then
  suspend <= 0;
end if;
end process read_proc;

write_proc : process
    -- Insert file-path
    file      out_file  : text is out
              "C:\Users\Tom\Desktop\out_image.txt";
    variable  out_line  : line;
begin
wait until clock = '0' and clock'event;
if(endoffile = '0') then
    if (suspend = 1) then
        for j in 0 to (3*WIDTH-1) loop
        if (j mod 3 = 0) then
            if (row < 2 or row > (HEIGHT-3)) then
                    if (row_order = 0) then
                        R_0((j/3)*2) <= row_0(j);
                        G_0((j/3)*2) <= row_0(j+1);
                        B_0((j/3)*2) <= row_0(j+2);
                        R_1((j/3)*2) <= row_0(j);
```

```
                G_1((j/3)*2) <= row_0(j+1);
                B_1((j/3)*2) <= row_0(j+2);
                R_0(((j/3)*2)+1) <= row_0(j);
                G_0(((j/3)*2)+1) <= row_0(j+1);
                B_0(((j/3)*2)+1) <= row_0(j+2);
                R_1(((j/3)*2)+1) <= row_0(j);
                G_1(((j/3)*2)+1) <= row_0(j+1);
                B_1(((j/3)*2)+1) <= row_0(j+2);
            elsif(row_order = 1) then
                R_0((j/3)*2) <= row_1(j);
                G_0((j/3)*2) <= row_1(j+1);
                B_0((j/3)*2) <= row_1(j+2);
                R_1((j/3)*2) <= row_1(j);
                G_1((j/3)*2) <= row_1(j+1);
                B_1((j/3)*2) <= row_1(j+2);
                R_0(((j/3)*2)+1) <= row_1(j);
                G_0(((j/3)*2)+1) <= row_1(j+1);
                B_0(((j/3)*2)+1) <= row_1(j+2);
                R_1(((j/3)*2)+1) <= row_1(j);
                G_1(((j/3)*2)+1) <= row_1(j+1);
                B_1(((j/3)*2)+1) <= row_1(j+2);
            elsif(row_order = 2) then
                R_0((j/3)*2) <= row_2(j);
                G_0((j/3)*2) <= row_2(j+1);
                B_0((j/3)*2) <= row_2(j+2);
                R_1((j/3)*2) <= row_2(j);
                G_1((j/3)*2) <= row_2(j+1);
                B_1((j/3)*2) <= row_2(j+2);
                R_0(((j/3)*2)+1) <= row_2(j);
                G_0(((j/3)*2)+1) <= row_2(j+1);
                B_0(((j/3)*2)+1) <= row_2(j+2);
                R_1(((j/3)*2)+1) <= row_2(j);
                G_1(((j/3)*2)+1) <= row_2(j+1);
                B_1(((j/3)*2)+1) <= row_2(j+2);
            end if;
        elsif (j < 6 or j > (3*WIDTH-7)) then
            if (row_order = 0) then
                R_0((j/3)*2) <= row_0(j);
```

```
            G_0((j/3)*2) <= row_0(j+1);
            B_0((j/3)*2) <= row_0(j+2);
            R_1((j/3)*2) <= row_0(j);
            G_1((j/3)*2) <= row_0(j+1);
            B_1((j/3)*2) <= row_0(j+2);
            R_0(((j/3)*2)+1) <= row_0(j);
            G_0(((j/3)*2)+1) <= row_0(j+1);
            B_0(((j/3)*2)+1) <= row_0(j+2);
            R_1(((j/3)*2)+1) <= row_0(j);
            G_1(((j/3)*2)+1) <= row_0(j+1);
            B_1(((j/3)*2)+1) <= row_0(j+2);
        elsif(row_order = 1) then
            R_0((j/3)*2) <= row_1(j);
            G_0((j/3)*2) <= row_1(j+1);
            B_0((j/3)*2) <= row_1(j+2);
            R_1((j/3)*2) <= row_1(j);
            G_1((j/3)*2) <= row_1(j+1);
            B_1((j/3)*2) <= row_1(j+2);
            R_0(((j/3)*2)+1) <= row_1(j);
            G_0(((j/3)*2)+1) <= row_1(j+1);
            B_0(((j/3)*2)+1) <= row_1(j+2);
            R_1(((j/3)*2)+1) <= row_1(j);
            G_1(((j/3)*2)+1) <= row_1(j+1);
            B_1(((j/3)*2)+1) <= row_1(j+2);
        elsif(row_order = 2) then
            R_0((j/3)*2) <= row_2(j);
            G_0((j/3)*2) <= row_2(j+1);
            B_0((j/3)*2) <= row_2(j+2);
            R_1((j/3)*2) <= row_2(j);
            G_1((j/3)*2) <= row_2(j+1);
            B_1((j/3)*2) <= row_2(j+2);
            R_0(((j/3)*2)+1) <= row_2(j);
            G_0(((j/3)*2)+1) <= row_2(j+1);
            B_0(((j/3)*2)+1) <= row_2(j+2);
            R_1(((j/3)*2)+1) <= row_2(j);
            G_1(((j/3)*2)+1) <= row_2(j+1);
            B_1(((j/3)*2)+1) <= row_2(j+2);
        end if;
```

```
            else
                 if (row_order = 0) then
                   -- Row 1
                   -- Row 2
                   -- Row 0
                    if (abs(row_2(j-3) - row_2(j+3)) < 10 and
                    abs(row_0(j) - row_1(j)) < 10) then
                       if (abs(row_1(j) - row_2(j-3)) < 10) then
                           R_0((j/3)*2) <= row_2(j-3);
                           G_0((j/3)*2) <= row_2(j-2);
                           B_0((j/3)*2) <= row_2(j-1);
                       else
                           R_0((j/3)*2) <= row_2(j);
                           G_0((j/3)*2) <= row_2(j+1);
                           B_0((j/3)*2) <= row_2(j+2);
                       end if;
                       if (abs(row_1(j) - row_2(j+3)) < 10) then
                           R_0((j/3)*2+1) <= row_2(j+3);
                           G_0((j/3)*2+1) <= row_2(j+4);
                           B_0((j/3)*2+1) <= row_2(j+5);
                       else
                           R_0((j/3)*2+1) <= row_2(j);
                           G_0((j/3)*2+1) <= row_2(j+1);
                           B_0((j/3)*2+1) <= row_2(j+2);
                       end if;
                       if (abs(row_2(j-3) - row_0(j)) < 10) then
                           R_1((j/3)*2) <= row_2(j-3);
                           G_1((j/3)*2) <= row_2(j-2);
                           B_1((j/3)*2) <= row_2(j-1);
                       else
                           R_1((j/3)*2) <= row_2(j);
                           G_1((j/3)*2) <= row_2(j+1);
                           B_1((j/3)*2) <= row_2(j+2);
                       end if;
                       if (abs (row_0(j) - row_2(j+3)) < 10) then
                           R_1((j/3)*2+1) <= row_2(j+3);
                           G_1((j/3)*2+1) <= row_2(j+4);
                           B_1((j/3)*2+1) <= row_2(j+5);
```

```
                    else
                        R_1((j/3)*2+1) <= row_2(j);
                        G_1((j/3)*2+1) <= row_2(j+1);
                        B_1((j/3)*2+1) <= row_2(j+2);
                    end if;
                else
                    R_0((j/3)*2) <= row_2(j);
                    G_0((j/3)*2) <= row_2(j+1);
                    B_0((j/3)*2) <= row_2(j+2);
                    R_0((j/3)*2+1) <= row_2(j);
                    G_0((j/3)*2+1) <= row_2(j+1);
                    B_0((j/3)*2+1) <= row_2(j+2);
                    R_1((j/3)*2) <= row_2(j);
                    G_1((j/3)*2) <= row_2(j+1);
                    B_1((j/3)*2) <= row_2(j+2);
                    R_1((j/3)*2+1) <= row_2(j);
                    G_1((j/3)*2+1) <= row_2(j+1);
                    B_1((j/3)*2+1) <= row_2(j+2);
                end if;
            elsif(row_order = 1) then
                -- Row 2
                -- Row 0
                -- Row 1
                 if (abs(row_0(j-3) - row_0(j+3)) < 10 and
                 abs(row_1(j) - row_2(j)) < 10) then
                    if (abs(row_2(j) - row_0(j-3)) < 10) then
                        R_0((j/3)*2) <= row_0(j-3);
                        G_0((j/3)*2) <= row_0(j-2);
                        B_0((j/3)*2) <= row_0(j-1);
                    else
                        R_0((j/3)*2) <= row_0(j);
                        G_0((j/3)*2) <= row_0(j+1);
                        B_0((j/3)*2) <= row_0(j+2);
                    end if;
                    if (abs(row_2(j) - row_0(j+3)) < 10) then
                        R_0((j/3)*2+1) <= row_0(j+3);
                        G_0((j/3)*2+1) <= row_0(j+4);
                        B_0((j/3)*2+1) <= row_0(j+5);
```

```
                else
                    R_0((j/3)*2+1) <= row_0(j);
                    G_0((j/3)*2+1) <= row_0(j+1);
                    B_0((j/3)*2+1) <= row_0(j+2);
                end if;
                if (abs(row_0(j-3) - row_1(j)) < 10) then
                    R_1((j/3)*2) <= row_0(j-3);
                    G_1((j/3)*2) <= row_0(j-2);
                    B_1((j/3)*2) <= row_0(j-1);
                else
                    R_1((j/3)*2) <= row_0(j);
                    G_1((j/3)*2) <= row_0(j+1);
                    B_1((j/3)*2) <= row_0(j+2);
                end if;
                if (abs(row_1(j) - row_0(j+3)) < 10) then
                    R_1((j/3)*2+1) <= row_0(j+3);
                    G_1((j/3)*2+1) <= row_0(j+4);
                    B_1((j/3)*2+1) <= row_0(j+5);
                else
                    R_1((j/3)*2+1) <= row_0(j);
                    G_1((j/3)*2+1) <= row_0(j+1);
                    B_1((j/3)*2+1) <= row_0(j+2);
                end if;
            else
                R_0((j/3)*2) <= row_0(j);
                G_0((j/3)*2) <= row_0(j+1);
                B_0((j/3)*2) <= row_0(j+2);
                R_0((j/3)*2+1) <= row_0(j);
                G_0((j/3)*2+1) <= row_0(j+1);
                B_0((j/3)*2+1) <= row_0(j+2);
                R_1((j/3)*2) <= row_0(j);
                G_1((j/3)*2) <= row_0(j+1);
                B_1((j/3)*2) <= row_0(j+2);
                R_1((j/3)*2+1) <= row_0(j);
                G_1((j/3)*2+1) <= row_0(j+1);
                B_1((j/3)*2+1) <= row_0(j+2);
            end if;
        elsif(row_order = 2) then
```

```
-- Row 0
-- Row 1
-- Row 2
 if (abs(row_1(j-3) - row_1(j+3)) < 10 and
 abs(row_0(j) - row_2(j)) < 10) then
    if (abs(row_0(j) - row_1(j-3)) < 10) then
        R_0((j/3)*2) <= row_1(j-3);
        G_0((j/3)*2) <= row_1(j-2);
        B_0((j/3)*2) <= row_1(j-1);
    else
        R_0((j/3)*2) <= row_1(j);
        G_0((j/3)*2) <= row_1(j+1);
        B_0((j/3)*2) <= row_1(j+2);
    end if;
    if (abs(row_0(j) - row_1(j+3)) < 10) then
        R_0((j/3)*2+1) <= row_1(j+3);
        G_0((j/3)*2+1) <= row_1(j+4);
        B_0((j/3)*2+1) <= row_1(j+5);
    else
        R_0((j/3)*2+1) <= row_1(j);
        G_0((j/3)*2+1) <= row_1(j+1);
        B_0((j/3)*2+1) <= row_1(j+2);
    end if;
    if (abs(row_1(j-3) - row_2(j)) < 10) then
        R_1((j/3)*2) <= row_1(j-3);
        G_1((j/3)*2) <= row_1(j-2);
        B_1((j/3)*2) <= row_1(j-1);
    else
        R_1((j/3)*2) <= row_1(j);
        G_1((j/3)*2) <= row_1(j+1);
        B_1((j/3)*2) <= row_1(j+2);
    end if;
    if (abs(row_2(j) - row_1(j+3)) < 10) then
        R_1((j/3)*2+1) <= row_1(j+3);
        G_1((j/3)*2+1) <= row_1(j+4);
        B_1((j/3)*2+1) <= row_1(j+5);
    else
        R_1((j/3)*2+1) <= row_1(j);
```

```
                        G_1((j/3)*2+1) <= row_1(j+1);
                        B_1((j/3)*2+1) <= row_1(j+2);
                    end if;
                else
                    R_0((j/3)*2) <= row_1(j);
                    G_0((j/3)*2) <= row_1(j+1);
                    B_0((j/3)*2) <= row_1(j+2);
                    R_0((j/3)*2+1) <= row_1(j);
                    G_0((j/3)*2+1) <= row_1(j+1);
                    B_0((j/3)*2+1) <= row_1(j+2);
                    R_1((j/3)*2) <= row_1(j);
                    G_1((j/3)*2) <= row_1(j+1);
                    B_1((j/3)*2) <= row_1(j+2);
                    R_1((j/3)*2+1) <= row_1(j);
                    G_1((j/3)*2+1) <= row_1(j+1);
                    B_1((j/3)*2+1) <= row_1(j+2);
                end if;
            end if;
        end if;
    end if;
end loop;
for i in 0 to 1 loop
    if (i = 0) then
        for j in 0 to (2*WIDTH-1) loop
            write(outline, R_0(j));
            writeline(outfile, outline);
            write(outline, G_0(j));
            writeline(outfile, outline);
            write(outline, B_0(j));
            writeline(outfile, outline);
        end loop;
    else
        for j in 0 to (2*WIDTH-1) loop
            write(outline, R_1(j));
            writeline(outfile, outline);
            write(outline, G_1(j));
            writeline(outfile, outline);
            write(outline, B_1(j));
```

```
                writeline(outfile, outline);
            end loop;
        end if;
    end loop;
    cont_read <= 1;
    else
        cont_read <= 0;
    end if;
else
null;
end if;
end process write_proc;
end Behavioral;
```