Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Thomas Stenseth

FPGA based video scaling for broadcast systems

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Kjetil Svarstad
June 2019

NTNU
Kunnskap for en bedre verden

Thomas Stenseth

FPGA based video scaling for broadcast systems

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Kjetil Svarstad
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Problem Description

Assignment title: FPGA based video scaling for broadcast systems

Implement FPGA based video scaler with support for upscaling and downscaling. The scaler should be build-time configurable for scaling between any video resolution. It should also support switching between preconfigured scaling modes during runtime.

Example of preconfigured video scaling modes:

- 720p to 1080p @ 60fps
- 1080p to 720p @ 60fps
- 1080p to 2160p @ 60fps
- 2160p to 1080p @ 60fps

Analyze the performance of the different scaling methods with focus on resource usage, latency and maximum frequency. Find a recommendation for the optimal solution for broadcast video systems, by comparing the different scaling filter implementations. Where the optimal solution offers the best tradeoff between scaled video quality and FPGA resource usage.

The finished system should be verified and documented using best practice methods.

Assignment given: 15. January 2019

Supervisor: Professor Kjetil Svarstad, IES NTNU

Co-Supervisor: Lars Erik Songe Paulsen, Appear TV

Summary

Video scaling is a process used to change the resolution of a video. This is widely used in broadcast systems to be able to support multiple devices using different resolution. Since broadcast systems relies on a continuous non-faulty operation, a hardware-based implementation is preferred over a software-based one, as this often has better performance and stability. The focus on this thesis is to implement a hardware-based video scaler, with a focus on resource usage, latency and maximum frequency, and to verify the implemented design using best practice methods.

A scaler with support for nearest-neighbor and bilinear interpolation was implemented in this thesis. The results from the synthesis test shows that the design meets its performance requirements. Both interpolation methods were able to operate at a maximum frequency above 300 MHz, while only using 1% of the resources of an Intel Arria 10 FPGA, and with a 4-line framebuffer the latency is four lines of pixels.

The objective image quality test shows lower performance of the implemented scaler algorithms compared to the reference algorithms, however the subjective test shows little or no difference, raising the suspicion that the lower objective results are a result of shifting in pixel positions, not incorrect pixel color values.

UVVM was used to verify the sub-modules of the design, and an Avalon-ST VIP was implemented for this purpose. Unfortunately, due to time limitations, the top level of the design were not completed, preventing the use of the Avalon-ST VIP for verification of the scaler algorithms.

With completion of the top level of the design, this video scaler would be able to handle scaling of video up to a resolution of 4k with a 30Hz framerate.

Sammendrag

Video skalering er en prosess brukt for å endre oppløsningen på en video. Dette er mye brukt i profesjonelle kringkastningssystem som skal støtte mange forskjellige typer brukerutstyr med ulik oppløsning. Siden profesjonelt kringkastningsutstyr krever å kunne operere med en kontinuerlig driftstid uten feil på systemet, er en maskinvare-basert løsning å foretrekke over en programvare-basert, da denne ofte har en høyere ytelse og bedre stabilitet. Denne masteroppgaven fokuserer på å designe og implementere en maskinvare-basert video skaleringskrets, og verifisere denne. Designmålet er å kunne kjøre kretsen på så høy frekvens med så lav forsinkelse som mulig, mens samtidig å kunne ha et lavt ressursbruk.

I denne masteroppgaven har en skalererkrets som støtter nærmeste-nabo og bi-lineær interpolasjon blitt designet og implementert. Resultatene fra syntesetestingen viste at skaleringskretsen oppfylte spesifikasjonene som var satt. Den opererte med en maksimumsfrekvens på over 300 MHz for begge interpolasjonsalgoritmene, mens den kun brukte 1% av ressursene som var tilgjengelig på en Intel Arria 10 FPGA. Med et 4-linjers bufferminne er forsinkelsen gjennom kretsen fire piksellinjer.

Den objektive bildekvalitetstesten ga et lavere resultat for den maskinvare-baserte skalereren sammenlignet med referansealgoritmene. Derimot kunne man fra de subjektive testene ikke se noen tydelig forskjell på resultatene sammenlignet med referansen. Dette hinter til en mulighet for at de dårlige objektive resultatene kom fra at pikslene ble flyttet på i posisjon, og ikke at pikslene i seg selv hadde feil fargeverdi.

UVVM ble brukt til å verifisere under-modulene i designet, og en Avalon-ST VIP ble implementert for dette formålet. Dessverre på grunn av tidsmangel, ble ikke toppnivået av designet fullført, noe som gjorde at man ikke kunne bruke UVVM til å verifisere skaleringsalgoritmene.

Ut ifra resultatene kunne man konkludere med at hadde toppnivået på skaleringskretsen blitt fullført, så kunne dette designet ha skalert video med opp til 4k oppløsning og 30Hz bildefrekvens.

Preface

This report is the result of a Master's thesis conducted during the spring of 2019. It is the conclusion of a five year study in Electronics Engineering at the Department of Electronic Systems at the Norwegian University of Science and Technology.

A lot of work has been put down into this thesis, and it has given me a lot of experience in digital design and VHDL coding techniques. The implementation of the Avalon-ST VIP for UVVM were a big challenge, but it gave me good insight into verification methods for VHDL designs.

I would like to thank my supervisor Professor Kjetil Svarstad at NTNU, and my co-supervisor Lars Erik Sogne Paulsen at Appear TV for their help and guidance during this project. I would also like to thank Appear TV for providing me with the necessary software needed to implement, test and verify this design.

June 10, 2019

Thomas Stenseth

Table of Contents

Problem Description i

Summary iii

Sammendrag v

Preface vii

Table of Contents xi

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Background 1
1.2 Objectives 2
1.3 Approach and limitations 2
1.4 Features and contributions 3
1.5 Report Outline 3

2 Video Scaling 5

2.1 Basics of Video Scaling 5
2.2 Nearest-Neighbor Interpolation 6
2.3 Bilinear Interpolation 7
2.4 Bicubic Interpolation 9

3 Interpolation on an FPGA 11

3.1 Reverse Mapping 11
3.2 Framebuffer 12
3.3 Nearest-Neighbor Interpolation 14
3.4 Bilinear Interpolation 15

4 Hardware Implementation 17

4.1 Fixed-Point Numbers 17

4.2 Top Level Design . . . 17
4.3 Interface . . . 18
4.3.1 Packet Transmission . . . 19
4.4 Controller . . . 20
4.4.1 Controller FSM . . . 20
4.5 Frame buffer . . . 21
4.5.1 Simple dual-port RAM . . . 21
4.5.2 Multi-port RAM . . . 22
4.5.3 An Improved Memory Configuration . . . 23
4.6 Scaler . . . 24
4.6.1 Scaler FSM . . . 24
4.6.2 Reverse Mapping . . . 25
4.6.3 Nearest-Neighbor Interpolation . . . 26
4.6.4 Bilinear Interpolation . . . 27
4.7 FIFOs . . . 29

5 Verification IP Implementation 31
5.1 UVVM . . . 31
5.1.1 Utility Library . . . 31
5.1.2 BFM (Bus Functional Model) . . . 32
5.1.3 VVC (VHDL Verification Component) . . . 32
5.2 Avalon-ST VIP . . . 33
5.2.1 Avalon-ST BFM . . . 33
5.2.2 Avalon-ST VVC . . . 34
5.2.3 Memory Concerns . . . 35

6 Testing and Verification Strategy 37
6.1 Verifying VHDL Modules . . . 37
6.2 Scaler Verification and Image Quality . . . 37
6.2.1 Matlab binary conversion . . . 38
6.2.2 Objective image quality models . . . 38
6.2.3 Test Images . . . 39
6.3 Synthesis test . . . 40

7 Results and Discussion 41
7.1 Verification of Sub-Modules . . . 41
7.1.1 Controller . . . 41
7.2 Scaler Verification and Image Quality . . . 42
7.2.1 Nearest-Neighbor Functional Verification . . . 43
7.2.2 Nearest-Neighbor Image Quality . . . 43
7.2.3 Bilinear Functional Verification . . . 45
7.2.4 Bilinear Image Quality . . . 46
7.2.5 Subjective image quality . . . 47
7.3 Synthesis test . . . 49

8 Conclusion and Future Work 53

8.1 Video Scaler 53
8.2 Avalon-ST VIP 54
8.3 Future Work 54

Bibliography **55**

Appendix **57**

A VHDL source code **57**
A.1 FIFO 57
A.2 Simple Dual-Port RAM 60
A.3 Multiport RAM 61
A.4 My Fixed Package 63
A.5 Nearest-Neighbor Scaling 64
A.6 Bilinear Scaling 4-line Framebuffer 69
A.7 Bilinear Scaling Full Size Framebuffer 77
A.8 Scaler Controller 83
A.9 Scaler Top Level 85

B VHDL Testbenches **89**
B.1 FIFO Testbench 89
B.2 Simple Dual-Port RAM Testbench 92
B.3 Multiport RAM Testbench 94
B.4 Scaler Algorithm Testbench With File IO 97
B.5 Scaler Top Level Testbench With UVVM 100
B.6 Scaler Top Level Testharness With UVVM 104

C Avalon-ST Verification IP source code **107**
C.1 Avalon-ST BFM 107
C.2 Avalon-ST VVC Testbench 117
C.3 Avalon-ST VVC Testharness 119

D MATLAB source code **121**
D.1 Image to Binary Function 121
D.2 Binary to Image Function 122

E Complete test results **123**
E.1 Nearest-Neighbor Interpolation 123
E.2 Bilinear Interpolation 125

List of Figures

2.1 Reverse mapping 6
2.2 Nearest neighbor interpolation 7
2.3 Bilinear interpolation 7
2.4 Bicubic interpolation 9

3.1 Serial video data 13
3.2 First x-value as the function of scale factor 14

4.1 Scaler top level 18
4.2 Timing diagram of packet transmission 19
4.3 Controller FSM 20
4.4 Simple dual-port RAM 22
4.5 Mapping pixels in source image to memories [1]. 23
4.6 Upscaler FSM 24

5.1 Structured VVC architecture [2] 33

6.1 Animated content used for testing 39
6.2 Natural content used for testing. Taken from Planet Earth II [3] 40

7.1 Output image from nearest-neighbor upscaling from 720p to 1080p 43
7.2 Nearest neighbor upscaling to 1080p using animated content 43
7.3 Nearest neighbor upscaling to 1080p using natural content 44
7.4 Output image from bilinear upscaling from 720p to 1080p 45
7.5 Bilinear bug using 4-line framebuffer 45
7.6 Bilinear upscaling to 1080p using animated content 46
7.7 Bilinear upscaling to 1080p using natural content 47
7.8 MATLAB vs VHDL bilinear upscaling from 360p to 1080p using animated content 48
7.9 MATLAB vs VHDL bilinear upscaling from 360p to 1080p using natural content 48
7.10 Synthesis test nearest neighbor DSP pipeline 51
7.11 Synthesis test bilinear DSP pipeline 51

List of Tables

5.1 Writing data to FIFO from testbench using BFM 32

7.1 Synthesis test of sub modules 49
7.2 Synthesis test using Intel Quartus 18.1 50
7.3 Performance on Intel Arria 10 GX 1150 50
7.4 Resource usage on Intel Arria 10 GX 1150 52

E.1 MATLAB nearest-neighbor, animated content 123
E.2 VHDL nearest-neighbor, animated content 123
E.3 MATLAB nearest-neighbor, natural content 124
E.4 VHDL nearest-neighbor, natural content 124
E.5 MATLAB bilinear, animated content 125
E.6 VHDL bilinear, animated content 125
E.7 MATLAB bilinear, natural content 125
E.8 VHDL bilinear, natural content 126

Chapter 1

Introduction

This master thesis builds upon an earlier project thesis conducted at NTNU in the fall of 2018 [4]. In the project thesis, the most common algorithms used for scaling video were explored, and an implementation of these were done in MATLAB. They were compared with each other with respect to ease of implementation, complexity of the algorithm, and their respective performance in image quality after scaling. This master thesis is thus the final step in implementing these algorithms in hardware, and building a working system that performs video scaling.

As this thesis heavily explores hardware implementation of mathematical algorithms, the reader is expected to have knowledge in mathematics, FPGA programming, hardware design and verification methods. Preferably the reader should also have some knowledge about digital video, and be familiar with the concept of color spaces, data streams, and how digital video is being presented to the user.

1.1 Background

As current video formats are evolving and people are moving away from watching linear TV to watching content on multiple devices, broadcast equipment that supports these new formats is required. One property of these new devices is that they have displays with many different resolutions and aspect ratios. It is therefore necessary to scale video to support these new formats.

Scaling of video requires scaling algorithms which reconstruct data that is not present in the original material. This is a computational heavy workload, and the design and implementation of the scaling algorithm is key for the end result. Using pure software-based scalers to perform the scaling operation is heavily dependent on the underlying hardware, and the other tasks performed by this hardware simultaneously. Because of this, software-based solutions is not the most reliable and best performing solutions in professional broadcast equipment.

A hardware implementation is preferred over a software-based one in broadcasting systems that require stable 24/7 operation. This way the system is guaranteed to meet its specifications, and the performance is stable over longer periods without the concerns for which other tasks that are running simultaneously. Given that new standards for video delivery are continuously being developed as well, it is often desirable to be able to update the current systems to support these new features. A hardware implementation on an FPGA is therefore a good choice, as this both guarantees the performance demands to be met, as well as having the possibility to be updated with newer video standards as these are being released.

1.2 Objectives

This report will focus on the two most used scaling algorithms, and implement these on an FPGA using VHDL. These two algorithms will be compared with regards to how much resources they use on an FPGA, how well they perform in terms of speed, and the image quality of the output produced. To verify the correctness of the system, UVVM (Universal VHDL Verification Methodology) will be used, and a Verification IP that complies with the Avalon-ST Video interface will be implemented for this purpose.

In summary this project aims to:

- Implement a video scaler in VHDL supporting upscaling between resolutions commonly used in broadcast systems.
- Implement a VIP (Verification IP) for the Avalon-ST Video interface compliant with the open-source UVVM (Universal VHDL Verification Methodology) library.
- Test and verify the implemented modules with a focus on meeting the quality and performance requirements set for the system.

1.3 Approach and limitations

Video scaling can be performed either as an up- or downscaling process. In the downscaling process the output consists of less data than the original input, and thus this is a reduction of information. For the upscaling process the opposite is true, and the system has to generate more information than what was present in the original source material. This is a much more heavy computational workload, and in many cases this will be the limitation for how well the system performs. This thesis will therefore focus on the upscaling process in regards of the implementation and testing.

In order to properly test the design, the testing strategy was split into two parts. The first part focuses on functional verification of the signaling and state machines using UVVM and the Avalon-ST Verification IP implemented for this. The other part focuses on correctness of the video output generated by the scaler. This was tested by creating a MATLAB

script that read the output data from the scaler, and compared this to the built-in scaler in MATLAB which served as a reference.

The Avalon-ST Video interface specification was chosen for this project because this design target the Intel Arria 10 FPGA family, and Avalon-ST Video is the interface used by Intels own proprietary IP cores. This makes the Avalon-ST Video interface well documented through Intels own user guides, see [5] and [6], and we can expect this interface to be well suited for implementation of a video scaler on an Intel Arria 10 FPGA.

1.4 Features and contributions

A major contribution of this project is the Avalon-ST VIP implemented for verification with UVVM. This VIP is non-existing amongst the included VIPs in UVVM, or withing the open-source UVVM community, and it will therefore be uploaded to the UVVM community with a MIT licence for public use. It can then serve as a foundation for other users who need UVVM verification of modules utilizing the Avalon-ST interface.

The main contribution will be the implemented scaling algorithm together with the submodules created for this project. These will also be published in a public repository on GitHub with a MIT licence for others to use. A lot of groundwork has been made to optimize these algorithms to utilize built-in DSPs on the FPGA, so these designs can be used as a basis for other video scaling implementations.

1.5 Report Outline

This report first presents the basics of video scaling together with a detailed explanation of the inner workings of the different scaling algorithms in Chapter 2. Chapter 3 then examines these algorithms further, and provide a possible manner to which these algorithms can be implemented on an FPGA. The actual hardware implementation details is discussed in Chapter 4, where the implementation details for each of the sub-modules making up this design is discussed separately. Chapter 5 gives a brief introduction to UVVM and its components, before discussing the implementation details of the Avalon-ST VIP. The testing and verification strategy is discussed in Chapter 6, and the test images used for this project are presented here. Finally the results are presented in Chapter 7 together with discussions around these, and the final conclusion is drawn in Chapter 8.

Chapter 2

Video Scaling

2.1 Basics of Video Scaling

Video scaling is the process of changing the resolution of the frames making up the video. If the resolution is reduced from its original size, a process know as downscaling, the amount of information in each frame will be reduced. This happens since the number of pixels in each frame is reduced. Upscaling, on the other hand, is the process of increasing the amount of pixels in each frame, which also increases the amount of information in each frame. Construction of new pixels in a scaling process from the pixels in the original frames is known as interpolation.

When a video frame is scaled using an interpolation algorithm, the pixels in the new frame has to be mapped to the original frame by a mapping function. This mapping function defines which pixels in the original frame that should be used by the interpolation algorithm to create the new pixel. There are two directions the mapping function could operate, and these are known as *source-to-target* and *target-to-source* mapping, or *forward-* and *reverse mapping* respectively.

In the forward mapping function, each pixel in the source frame is used as a basis and the corresponding pixel in the target frame is calculated from these. This can be described as the mathematical function

$$(x', y') = T(x, y) \tag{2.1}$$

where x' and y' is the target pixels. This is a hard way to compute the new pixel values, since the mapping function might not have all the necessary information about the target frame to produce a correct result. A result of this is the possibility of introducing holes in the target pixels when one or more pixels are forgotten [7]. A more natural way is therefore to use reverse mapping.

The reverse mapping function uses the new pixels as a basis, and calculate their respective position in the source frame. This can be seen in Figure 2.1.

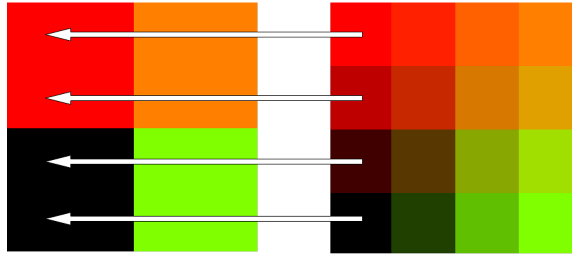


Figure 2.1: Reverse mapping

The mathematical expression of reverse mapping can be expressed as

$$(x, y) = T^{-1}(x', y') \tag{2.2}$$

and by using this mapping function, it is easier to determine which pixels in the source frame the interpolation algorithm should utilize to generate the new pixel in the target frame. By using reverse mapping you also ensure that all the pixels in the target frame are given values. In the forward mapping method there is a possibility of generating ”holes” which are pixels without intensity values. This is not the case with reverse mapping.

2.2 Nearest-Neighbor Interpolation

The most basic interpolation algorithm is the nearest-neighbor interpolation algorithm. After the reverse mapping function has calculated the new pixels relative position in the source frame, nearest-neighbor algorithm simply chooses the value of the pixel in the source frame that is closest to the new pixel. This can be described mathematically with a kernel using the weighting coefficients

$$W_{nn}(x, y) = \begin{cases} 1 & \text{for } -0.5 \leq x, y < 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

The function for calculating each pixel's intensity value is thereby given as

$$f(x', y') = f(x, y) \cdot W_{nn}(x' - x, y' - y) \tag{2.4}$$

An illustration of nearest-neighbor interpolation in a single dimension can be seen in Figure 2.2, where the black dots represent the original pixels in the source frame.

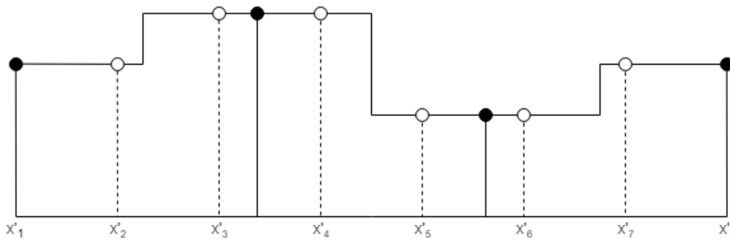


Figure 2.2: Nearest neighbor interpolation

Since nearest-neighbor interpolation simply takes the values of the nearest pixel, the end result is a very pixelated image. This is generally not a desired result, unless the source material is pixel art. However, nearest-neighbor interpolation has a very low computational requirement, and can easily be done in real-time as it is basically a copy-paste operation.

2.3 Bilinear Interpolation

Bilinear interpolation uses a 2×2 neighborhood of pixels, as a basis to assign appropriate intensity values to new pixels. It takes the 4 surrounding pixels of the position in the source frame calculated by the reverse mapping function, and calculate a weighted average of these to determine the new pixel intensity. Figure 2.3 demonstrates the concept of bilinear interpolation with (x', y') representing the position calculated by the reverse mapping function.

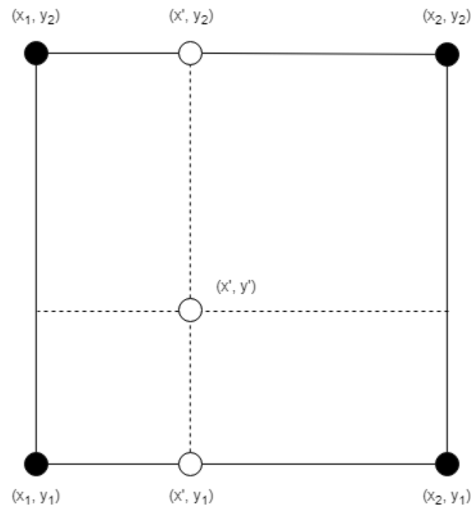


Figure 2.3: Bilinear interpolation

As the name suggest, bilinear interpolation uses a linear kernel to calculate the weight of

each pixel [8], and the weighting coefficients are given by

$$W_{lin}(x) = \begin{cases} 1 - x & \text{for } |x| < 1 \\ 0 & \text{for } |x| \geq 1 \end{cases}$$
$$W_{lin}(y) = \begin{cases} 1 - y & \text{for } |y| < 1 \\ 0 & \text{for } |y| \geq 1 \end{cases} \tag{2.5}$$

where each weighting coefficient determines intensity in one direction. The kernel for bilinear interpolation would then use the combined weighting coefficient

$$W_{bil}(x, y) = W_{lin}(x) \cdot W_{lin}(y) \tag{2.6}$$

Using the kernel with the weighting coefficients shown in Equation 2.5, we can develop a practical implementation-friendly algorithm for calculating the new pixel intensities. By first performing interpolation in the x-direction, we could generate two new pixel points, shown as (x', y_2) and (x', y_1) in Figure 2.3. These new points would then be used to calculate the final pixel intensity, shown as (x', y') in Figure 2.3.

By using the x-direction as the first interpolation direction, we get the two intermediate values (x', y_2) and (x', y_1) as given by Equation 2.7 and 2.8.

$$f(x', y_1) \approx \frac{x_2 - x'}{x_2 - x_1} f(x_1, y_1) + \frac{x' - x_1}{x_2 - x_1} f(x_2, y_1) \tag{2.7}$$

$$f(x', y_2) \approx \frac{x_2 - x'}{x_2 - x_1} f(x_1, y_2) + \frac{x' - x_1}{x_2 - x_1} f(x_2, y_2) \tag{2.8}$$

From Equation 2.7 and 2.8 we see that each of the four neighbouring pixels are counted towards the two new intermediate intensity values based on the distance they reside from the point given by the reverse mapping algorithm. These two intermediate values is then used to calculate the final value of the pixel. The two values are interpolated in the y-direction as shown in Equation 2.9

$$f(x', y') \approx \frac{y_2 - y'}{y_2 - y_1} f(x', y_1) + \frac{y' - y_1}{y_2 - y_1} f(x', y_2) \tag{2.9}$$

which yield the final intensity of the new pixel in the target frame.

By using a 2×2 neighborhood of pixels and taking a weighted sum of these, the end result of bilinear interpolation is a more correct result than nearest-neighbor interpolation. This way several pixels determine the value of the new pixel, and we avoid situations where the end result is very pixelated. This is especially favourable for natural content where sharp changes from pixel to pixel is scarce. However, bilinear interpolation is more computational demanding than nearest-neighbor as it requires several operations per pixel for the interpolation process.

2.4 Bicubic Interpolation

As with bilinear interpolation, bicubic interpolation uses a neighborhood of pixels as a basis to calculate the value of the new pixel. The neighborhood consist of 4×4 pixels, and an illustration of this can be seen in Figure 2.4.

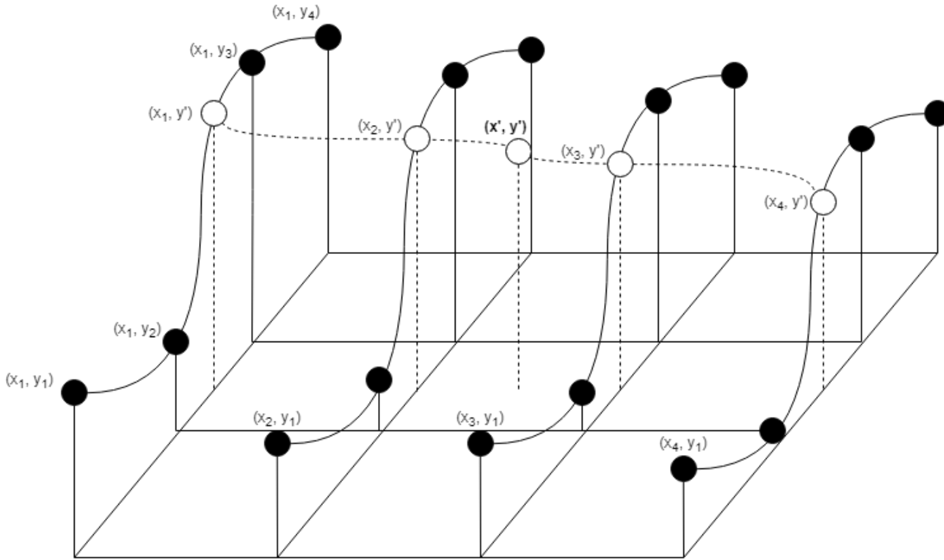


Figure 2.4: Bicubic interpolation

Bicubic interpolation utilizes a more advanced convolution algorithm to calculate the weight of the pixels from the source frame [9]. The weight of each pixel is calculated from the following kernel

$$W_{cub}(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \leq 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{if } 1 < |x| < 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

In this kernel the value of a is usually set to -0.5, as this is the only value that will achieve third-order precision [9]. Using $a = -0.5$ and having $|x| < 2$, which is the case for a 4×4 neighborhood, $W_{cub}(x)$ simplifies to

$$W_{cub}(x) = \begin{cases} 1.5|x|^3 - 2.5|x|^2 + 1 & \text{for } |x| \leq 1 \\ -0.5|x|^3 + 2.5|x|^2 - 4|x| + 2 & \text{if } 1 < |x| < 2 \end{cases} \tag{2.11}$$

The complete kernel for bicubic interpolation in two dimensions is given by multiplying the cubic kernel in both x- and y-direction:

$$W_{bic}(x, y) = W_{cub}(x) \cdot W_{cub}(y) \tag{2.12}$$

Using Figure 2.4 as a basis, and the kernel for bicubic interpolation to calculate the weight of each pixel in the 4×4 neighborhood, the formula for bicubic interpolation becomes the sum

$$f(x', y') = \sum_{x=\lfloor x' \rfloor - 1}^{\lfloor x' \rfloor + 2} \left[\sum_{y=\lfloor y' \rfloor - 1}^{\lfloor y' \rfloor + 2} f(x, y) \cdot W_{bic}(x' - x, y' - y) \right] \tag{2.13}$$

This operation is much more computational intensive than bilinear interpolation, but it can also easily be done in parallel. Because of the larger number of source pixels used to determine the value of the new pixel, and the third-order kernel, the end result of bicubic interpolation is usually more accurate and correct than for nearest-neighbor and bilinear interpolation [9].

Chapter 3

Interpolation on an FPGA

3.1 Reverse Mapping

Reverse mapping takes each pixel position in the output frame and calculates the corresponding pixel position in the input frame. This makes reverse mapping well suited for applications where the input can be buffered, and a streamed output is required, see chapter 9.2 in [7]. With reverse mapping you also ensure that there is no holes in the output frame, since the output frame is being generated based on the number of pixels in the output frame itself, not on the input frame.

In the previous project thesis [4], the reverse mapping function was handled by iterating though each pixel in the output frame, and calculate each pixels relative position in the input frame. This was done using the same method for coordinate orientation and interval calculation as was used by [8], and the equation is given as

$$
\begin{aligned}
x &= \frac{x'}{scalefactor} + 0.5 * \left(1 - \frac{1}{scalefactor}\right) \\
y &= \frac{y'}{scalefactor} + 0.5 * \left(1 - \frac{1}{scalefactor}\right)
\end{aligned} \tag{3.1}
$$

where (x, y) is the pixel position in the input frame and (x', y') is the pixel position in the output frame, with x and y being the column and row position respectively in a 2D matrix. The scale factor component is the number of pixels in one direction of the output frame compared to the input frame, so an upscaling from 1280x720 to 1920x1080 would give a scale factor of $1080/720 = 1.5$ in the x-direction. This would also be equal for the y-direction as the aspect ratio is preserved after scaling.

We can see from Equation 3.1 that this calculation has one multiplication and two division operations. Division operations on an FPGA is hard, especially when using non-integer

values as is the case with a scale factor of 1.5. Multiplication can be handled more easily with the use of built in DSPs on an FPGA. A more suited equation for calculating (x, y) on an FPGA would therefore be to rewrite Equation 3.1 to only have multiplications. This can be done by first separating out the scale factor division, which is a constant, as

$$c_1 = \frac{1}{scalefactor} \tag{3.2}$$

and then use this result to rewrite Equation 3.1 as

$$\begin{aligned} x &= x' * c_1 + 0.5 * (1 - c_1) \\ y &= y' * c_1 + 0.5 * (1 - c_1) \end{aligned} \tag{3.3}$$

Investigating Equation 3.3 we see that this can be further optimized. The final part of the equation is also always constant, and thus it is only the first part of the equation that needs to be calculated for each new (x', y') value. We could therefore separate the last part as

$$c_2 = 0.5 * (1 - c_1) \tag{3.4}$$

which would yield the final two equations to be performed on an FPGA for each pixel as

$$\begin{aligned} x &= x' * c_1 + c_2 \\ y &= y' * c_1 + c_2 \end{aligned} \tag{3.5}$$

Equation 3.5 only consist of one multiplication and one addition, this is something that can be well optimized on an FPGA by using DSPs and pipelining.

3.2 Framebuffer

To be able to use reverse mapping on an FPGA, the input video frame has to be buffered in a framebuffer. This is especially true for any interpolation algorithm that requires a neighborhood of pixels in an input frame to calculate the new pixel in the output frame. If the frame is not buffered, pixel values might be lost before the interpolation algorithm gets to use them for the calculation of the new pixel values.

In many cases a digital video signal is being delivered as a stream of serial data with one pixel being sent at a time, staring from the upper left-hand pixel of a video frame, and finishing with the lower right-hand pixel. Because of this, each pixel is only available for one clock cycle on the data bus, and thus is it very important that the pixel data is buffered in a framebuffer. An illustration of the serial video data transfer can be seen in Figure 3.1

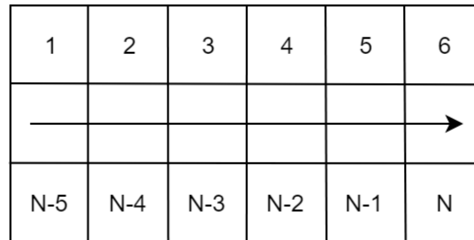


Figure 3.1: Serial video data

A complete frame of a video contains a lot of data. Given an input video with the resolution 1920x1080, each frame holds $\sim 2 \cdot 10^6$ pixels. If each pixel contains 24-bit of data, 8-bit per colour component, a complete frame consist of ~ 50 Mbit of data. This is a very large amount of data that needs to be stored in a framebuffer. Luckily we can exploit the fact that video data often is being transmitted in a series. This way, we only need to store some of the lines in the video frame at any given time.

Given that we use bilinear interpolation as our scaling algorithm, we need a neighborhood of 2×2 pixels to generate a new pixel for our output data. This way we only need two lines of video data in our framebuffer to do the interpolation. By using reverse mapping and starting the interpolation with the upper left-hand pixel in the output image, the scaling algorithm will flow naturally through the buffered lines in the framebuffer, at the same time generating output data in the same serial fashion as the input data. When the interpolation algorithm is done with these two lines, they are no longer needed, and their memory addresses in the framebuffer can be reused.

While the interpolation is taking place, new data will arrive on the input, and this data would need to be buffered in the framebuffer. If the scaler implementation produces one new video output line at the same rate as one input line is being received, the framebuffer will be emptied at the same rate as it is being filled. This way you would need a framebuffer that is twice the size of the minimum requirement of the pixel-neighborhood of the interpolation algorithm used.

Using bilinear interpolation as an example, this would require a 4-line framebuffer. When two lines have been processed, two new lines would have been filled in the framebuffer. When the interpolation algorithm starts working on these two new lines, the memory addresses of the two old ones are free, and they can be refilled by the incoming video data.

Theoretically, bilinear interpolation could also be performed using a 3-line framebuffer, or even a 2-line framebuffer where the new data is being written to the memory address just emptied by the scaler. However, this could lead to unwanted situations where data is overwritten before it is being read by the scaler if the scaler stalls for a couple of clock periods, or vice versa in a situation where the input data is being stalled while the scaler keeps on going. Using a framebuffer twice the size of the required neighborhood-pixels always put at least one line between the read and write process to the framebuffer. This way the design would have several clock cycles headroom to halt if for instance the input

data is interrupted for a couple of cycles.

3.3 Nearest-Neighbor Interpolation

As discussed in Chapter 2.2, using nearest-neighbor interpolation with reverse mapping consists of finding the pixel closest to the position in the source frame given by the reverse mapping function. Studying Equation 3.1 we see that using $scalefactor > 1$, which is the case for an upscaling process, c_2 from Equation 3.4 is always withing the boundaries

$$0 \leq c_2 \leq 0.5 \tag{3.6}$$

Looking at Equation 3.5 we also see that

$$0 \leq (x' * c_1) \leq 1 \tag{3.7}$$

when having $scalefactor > 1$, and that $(x' * c_1)$ decreases in value as c_2 increases. Knowing this we can make a plot of what the first x-value from Equation 3.5 would be when having different scale factors. This plot can be seen in Figure 3.2.

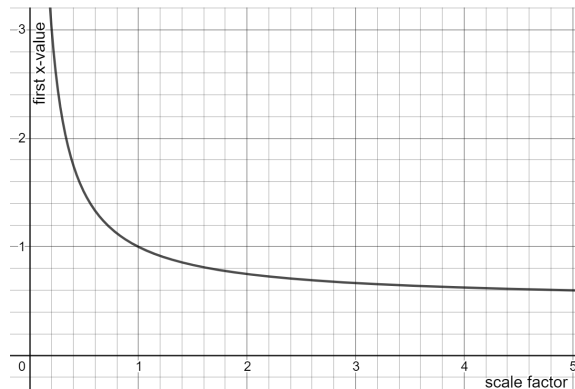


Figure 3.2: First x-value as the function of scale factor

As seen from the plot in Figure 3.2, having $scalefactor > 1$ always implies that

$$x < 1 \quad \textbf{when} \quad x' = 1. \tag{3.8}$$

The same is true for the first y-value. Knowing this is something that can be exploited in the nearest-neighbor interpolation algorithm. Memory addresses in VHDL are 0-indexed, which means that we can use a floor rounding method on the (x, y) values to do a nearest-neighbor interpolation. This means that the first pixel would be memory address 0, which

is the upper left-hand pixel in a framebuffer using the serial data as illustrated in Figure 3.1.

Continuation of the nearest-neighbor algorithm would then be to increase x' using the reverse mapping function. By doing this, and always use the floor function, the interpolation process would flow through one row of the input frame. When the row has been completed, the reverse mapping algorithm would increase y' by 1, and y would eventually take the value 1. To be able to calculate the memory addresses of the next row of pixels, we could use the known information about the resolution of the input video.

By knowing that data in the framebuffer is stored serially, we could increase the memory address to be read equal to the width of a row of pixels. This can be expressed with the equation

$$addr_{fb} = floor(y) * rx_w + floor(x) \tag{3.9}$$

where $addr_{fb}$ is the memory address in the framebuffer, and rx_w is the width of the input video frame. Thus by using Equation 3.5 to calculate the new pixels relative position in the input frame, increase x' to run through a row of pixels before increasing y', and then use Equation 3.9 to calculate the memory address in the framebuffer, we would have a working nearest-neighbor interpolation algorithm.

3.4 Bilinear Interpolation

As with nearest-neighbor, bilinear interpolation can use the same concept for calculating memory addresses in the framebuffer. Using the same concept for reverse mapping calculation, and the first address as given in Equation 3.9, we can extend this to get the addresses for the neighboring pixels as well. Since bilinear interpolation uses a neighborhood of 2×2 pixels, we can use Equation 3.9 to represent pixel (x_1, y_2) from Figure 2.3, and extend this address by one to get the pixel-data for pixel (x_2, y_2). This way these two pixels will have the memory addresses

$$\begin{aligned} (x_1, y_2) &= floor(y) * rx_w + floor(x) \\ (x_2, y_2) &= floor(y) * rx_w + floor(x) + 1 \end{aligned} \tag{3.10}$$

To get the two pixels (x_1, y_1) and (x_2, y_1) we can simply increase the memory address by the size of one row in the video frame. This way these two pixels would have the memory addresses

$$\begin{aligned} (x_1, y_1) &= (floor(y) + 1) * rx_w + floor(x) \\ (x_2, y_1) &= (floor(y) + 1) * rx_w + floor(x) + 1 \end{aligned} \tag{3.11}$$

One problem with these memory address calculation is that (x_2, y_2) will be outside the row on the final reverse mapping calculation, and (x_1, y_1) and (x_2, y_1) will be outside of the input video frame on the entire last row. However, this can be compensated by setting $x_2 = x_1$ for the last pixel in the row, and setting $y_1 = y_2$ for the final row.

After the framebuffer addresses have been calculated, the weighting of each of these pixels should be calculated. This is done by using Equations 2.7 and 2.8. Ideally this should be done using infinite precision, however, this is hard to do on an FPGA. Therefore a good option to allow for fast computation would be to use fixed-point numbers and DSPs to do these calculations. Finally the new pixel value is calculated by using Equation 2.9, and a rounding function to make the result an integer value.

The accuracy of this approach is decided by the number of bits used to represent the fractional part of these fixed-point numbers. Using the unsigned Q number format UQ*m.n*, the range and resolution of the numbers would be

$$\begin{aligned} \text{range: } & [0, 2^m - 2^{-n}] \\ \text{resolution: } & 2^{-n} \end{aligned} \tag{3.12}$$

where m is the number of bits representing the integer part, and n is the number of bits representing the fractional part. It is important to choose appropriate values for m and n so that as high a degree of accuracy is achieved, while being able to do calculations efficiently on the FPGAs built-in DSP.

Chapter 4

Hardware Implementation

4.1 Fixed-Point Numbers

This design were implemented with performance and resource usage in mind. The design goal was to be able to run the design at a minimum frequency of 300 MHz, and to use as little resources as possible on the FPGA.

As discussed in the previous chapter, DSP multiplication with fixed-point numbers could be a solution to meeting the frequency requirements. This is backed up by a white paper published by Xilinx, where the DSP performance using fixed-point representations were 16% faster with a 7.5x lower latency compared to using single-precision floating point numbers for a simple FIR filter implementation [10]. The resource usage was significant lower as well, with a 5x reduction in the number of DSPs required, and 11x lower LUT usage.

Looking at these numbers it is clear that fixed-point representation can have a great advantage over single-precision floating point numbers in designs that heavily utilizes DSPs, especially when it comes to resource usage, and this is the reason why fixed-point numbers were used in this design.

4.2 Top Level Design

The implementation of the scaler started with the planning of the top level design, and the different sub modules needed to perform the scaling operation. From Chapter 3 it was clear that a framebuffer was needed to buffer the input video data for the reverse mapping function to work properly. A controller was also needed to control the flow of video data to the framebuffer, and to start and stop the scaler itself according to the data present in the framebuffer. It therefore became clear that at least three main modules was needed, a controller, a framebuffer and a scaler. These three main modules can be seen in Figure 4.1.

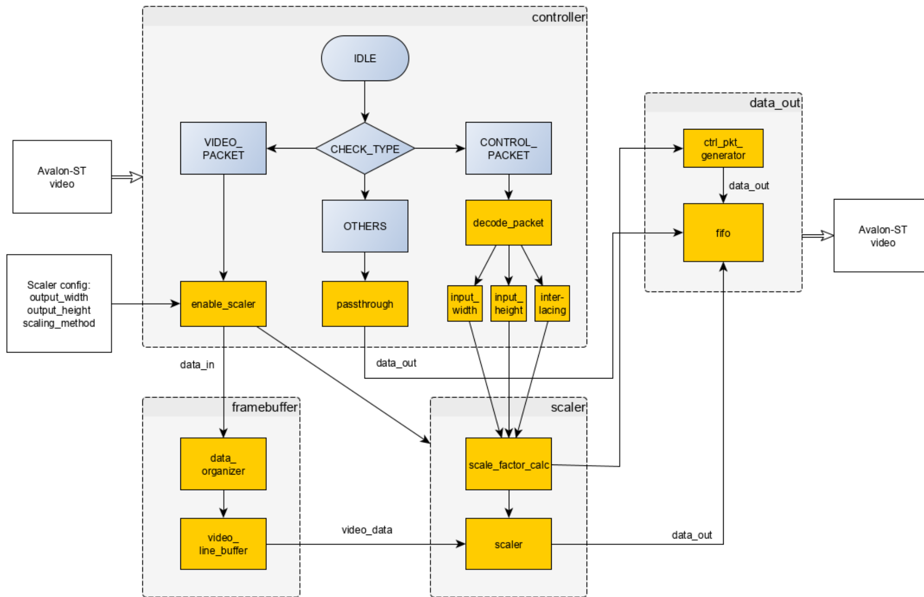


Figure 4.1: Scaler top level

As seen from Figure 4.1, the idea was to have the controller receiving all the input data and process this. When a scale operation was needed, the controller would start filling the framebuffer and subsequently enable the scaler when enough data had been filled in the framebuffer. The video data would then be processed by the scaler and passed on to an output FIFO that would deliver the finish data on the output of the scaler. The complete source code for the scaler implementation can be seen in Appendix A.

4.3 Interface

To be able to send and receive data to and from the scaler design, an main interface had to be specified. Since this project were targeted to run on an Intel Arria 10 FPGA, the most natural thing was to explore the interfaces used by Intels proprietary IP cores for video data, and see if a similar interface could be used in this design.

After studying the user guide on Intels Video and Image Processing Suite [6], it quickly became apparent that the Avalon-ST Video interface would be a suitable choice for this project. By using the same interface for this design, a lot of work would be saved by not having to test and validate the correctness of the interface itself, as one can expect this to be a good choice for transmitting video data as it is already used by Intels proprietary IP cores.

4.3.1 Packet Transmission

The choice of method for packet transmission in this design landed on the Avalon-ST YCbCr 4:4:4 Video Packet [6], only with a different position of the colour components in the data bus for an easier comparison with the old MATLAB implementation from the project thesis [4]. A timing diagram of this packet transmission method using 8-bit pixel values can be seen in Figure 4.2.

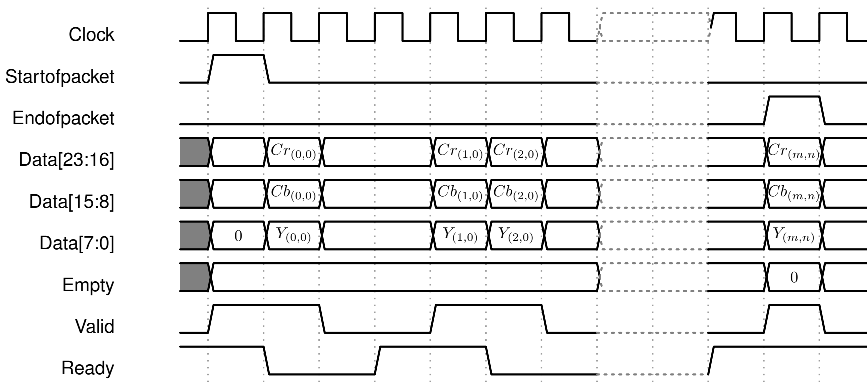


Figure 4.2: Timing diagram of packet transmission

As seen by Figure 4.2, this packet transmission specification uses a ”ready latency” of 1 clock cycle. This means that when a module asserts its ready signal, it will be ready to receive data on the next clock cycle. Likewise, when the ready signal is set to low, the module is expected to accept a packet coming on the next clock cycle. The valid signal is asserted to tell when there is data on the data bus.

Startofpacket and endofpacket signals are asserted to tell the module which symbol is the first and last respectively in a serial data transfer. This will later be used to tell when we have the beginning of a video frame, and when the last pixel in that video frame arrives.

From Figure 4.2 we can see that the startofpacket signal is asserted along with the data value ”0” on the clock cycle prior to the arrival of the first pixel data. This is because the first symbol transmitted is a packet type identifier telling the module what kind of data that is being transmitted. This design uses the same packet type identifiers as specified in the Avalon-ST Video transmission [6], hence the value ”0” represent a video data packet.

Finally the empty signal tells how many symbols in the last transmission there are which are not populated by pixel data. This is used when several pixels are transmitted in parallel. However, this design only uses a transmission of one pixel at a time, so the empty signal will not be used in this implementation.

4.4 Controller

The first part of the design that receives any data is the controller module from Figure 4.1. The main functionality of the controller module is to check what kind of data that is being sent to the scaler, and act accordingly. This is done by decoding the packet type identifiers sent with the Avalon-ST Video transmission.

There are two packets that are handled by the scaler, control packets and video data packets. Control packets contains information about the incoming video frame, like its resolution, while the video data packet contains pixel data from the incoming video frame. Other types of packages defined by the Avalon-ST Video transmission is not supported by this scaler design.

4.4.1 Controller FSM

A Mealy finite-state machine is used to control the controller with the packet type identifier being used as the input for the state machine. A flowchart of the state machine in the controller can be seen in Figure 4.3.

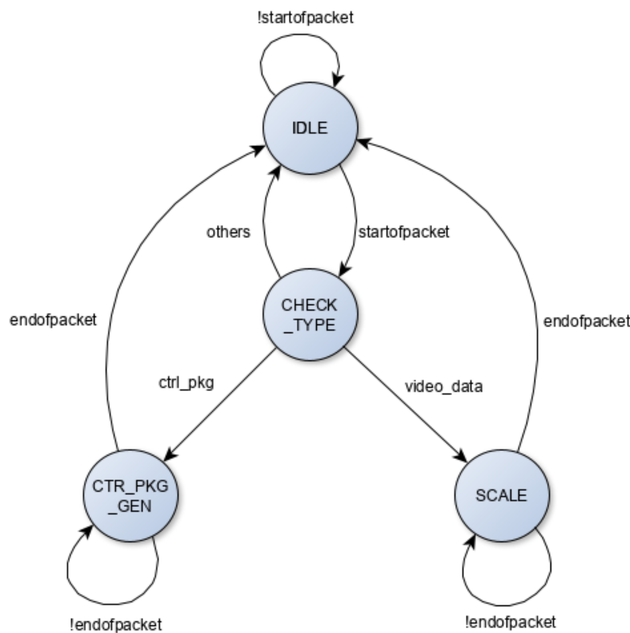


Figure 4.3: Controller FSM

From Figure 4.3 the controller starts in the IDLE state waiting for a startofpacket signal. When the controller receives a startofpacket signal it goes to the CHECK_TYPE state

which checks what kind of packet that is being received. If this is a non-supported packet type, the state machine returns to IDLE position and simply passes the packet through to the output of the scaler.

When the packet type is a control packet, the state machine goes to the CTR_PKG_GEN state where the control packet is decoded. The control packet contains information about the resolution of the upcoming video packet. This is passed on to the scaler to be able to set up the framebuffer to a correct size. The controller then generates a new control packet based on which output resolution the scaler is going to produce after scaling, and this is packed in the same way as the control packet described in ”Intel Avalon-ST Video Control Packets” [6]. Once the new control packet has been generated and sent on the output, the controller returns to the IDLE state, waiting for a new startofpacket.

In the case where the packet type is video data, the controller goes into its SCALE state. In this state a startofpacket signal is sent together with the first pixel data to the scaler, thereby initiating the filling of the framebuffer and starting the scaler. A big difference here in the internal design is that the startofpacket signal is sent to the scaling module together with the first pixel value $YCbCr_{(0,0)}$. The packet type identifier with its startofpacket signal from Figure 4.2 is not passed on to the scaler module as this is only used by the controller.

The controller holds the SCALE state until the endofpacket signal is received, in which the controller passes on the endofpacket signal together with the last data entry to the scaler, and returning to its IDLE state.

Receiving a reset signal always makes the controller go back to the IDLE state, and thereby aborting any current state.

4.5 Frame buffer

Once video data is received by the controller and passed on to the scaler module, it is buffered in a framebuffer. As discussed in Chapter 3.2, video data is received in a serial fashion starting with the upper left-hand pixel in an 2D image. This way several lines of pixels has to be buffered in the framebuffer before the scaler can begin.

4.5.1 Simple dual-port RAM

To be able to buffer a decent amount of data while still having good memory performance, a simple dual-port RAM using the built in M20K memory of the Arria 10 FPGA was implemented. An illustration of the simple dual-port RAM implemented can be seen in Figure 4.4.

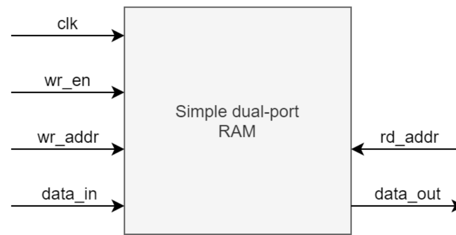


Figure 4.4: Simple dual-port RAM

The simple dual-port memory was implemented using a single process. At each rising edge of the clock, the data on the input would be written to the given memory address when wr_en was set to high as seen in Listing 1.

```
p_ram : process(clk_i)
begin
  if(rising_edge(clk_i)) then
     -- Write to RAM
     if(wr_en_i = '1') then
        ram_data(wr_addr_i) <= data_i;
     end if;
     -- Read from RAM
     ram_out     <= ram_data(rd_addr_i);
     ram_out_reg <= ram_out;
  end if;
end process p_ram;
```

Listing 1: Simple dual-port RAM

A register was used on the output of the simple dual-port RAM. This was done to achieve the highest possible performance from the RAM.

Nearest-neighbor interpolation was implemented using a 4-line framebuffer with simple dual-port RAM. This way the scaler design would have a lot of pixels buffered to be able to produce a steady output of pixels even if there would be an uneven flow of input video data. Using 8-bit pixel values and a 1080p video input, this would result in $4*1080\text{px}*24\text{-bit} \approx 104$ Kbit of memory being used on the FPGA.

4.5.2 Multi-port RAM

As discussed in Chapter 3.2, a 4-line framebuffer would be needed for bilinear interpolation. This was implemented for the nearest-neighbor interpolation algorithm using simple dual-port RAM. One problem with this implementation is that you are only able to read one value from the memory at a time, which becomes a problem in bilinear interpolation when you need 4 pixel values per calculation.

To overcome this a multi-port RAM design was implemented. This consisted of four identical simple dual-port RAM designs, where each sub-memory holds identical data.

Thus when there is new data on the input, and this is to be written to RAM, it is being written to all four simple dual-port RAMs. The output, however, is separate for each sub-memory. This allows the multi-port RAM to read four different pixel values each clock cycle.

The downside of this design is that it uses four times the memory by storing each pixel in four different memories. Because of this, the multi-port RAM takes up about $4 * 104 = 416$ Kbit of memory on the FPGA. However, using the Intel Arria 10 GX 1150, there are 54,260 Kbit of M20K memory available on the FPGA [11], so this only amounts to $\approx 1\%$ of the total memory available.

4.5.3 An Improved Memory Configuration

A better alternative to using four identical simple dual-port memories to store pixel data for bilinear interpolation could be to use what Suhaib A. Fahmy presented in [1], which can be seen in Figure 4.5.

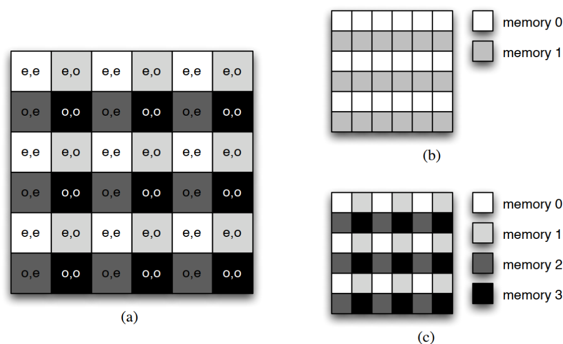


Figure 4.5: Mapping pixels in source image to memories [1].

In Fahmys paper you either use four simple dual-port memories, seen in Figure 4.5 (c), or two true dual-port memories, seen in Figure 4.5 (b), but the incoming pixels are stored in separate memories according to their position in the source image, seen in Figure 4.5 (a). This way you don't need to have four copies of every pixel and you save 3/4 of the memory used. However this requires a more complex logic to keep track of which pixels are in which memory. Because of this, and the time limiting factor, a simpler solution using four identical simple dual-port memories as framebuffer was implemented.

4.6 Scaler

The idea behind the scaler implementation was to have the controller dynamically setting up the scaler with respect to resolution and scaling algorithm based on the control packet received and a configuration file. This is seen in the top level design in Figure 4.1. Unfortunately, due to time limitations, this part of the design was never completed. The scaler module is thus currently working as an independent module using VHDL generics to control the input and output resolution, and some work is still left to connect til scaler module with the controller.

4.6.1 Scaler FSM

The scaler module is controlled by a Mealy finite-state machine, and a flowchart of this state machine can be seen in Figure 4.6.

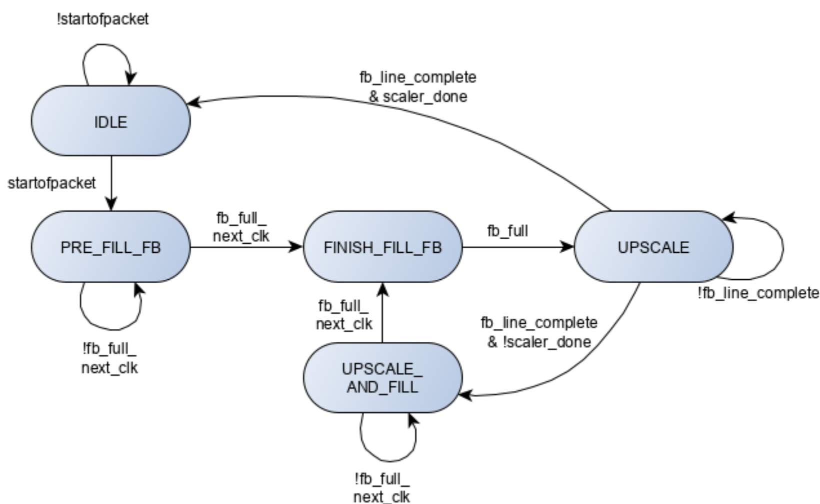


Figure 4.6: Upscaler FSM

When the scaler is in its IDLE state there is no data in the framebuffer. After receiving a startofpacket signal from the controller, the framebuffer is filled up with four lines of pixel data. When the framebuffer is full on the next clock cycle, it transitions to the FINISH_FILL_FB state. This is done because of the 1 clock cycle delay of the ready signal used in Avalon-ST Video [6], which requires the module to set its ready signal to low one clock cycle before receiving the last data. The ready signal goes low when the last pixel is filled in the framebuffer, and then the scaler goes to its UPSCALE state.

The upscaling takes place until one line in the framebuffer have been competed. The scaler runs through one line at the time due to the reverse mapping function, and when this line

is done, the framebuffer is ready to receive one new line of pixels. A transitioning is then happening to the UPSCALE_AND_FILL process, where both filling of that one line in the framebuffer and upscaling is taking place. The ready signal goes high at this point allowing the filling of this one line in the framebuffer, before going low in the FINISH_FILL_FB state to handle the 1 clock cycle delay.

When the reverse mapping function have run though all of the lines in the input video frame, the scaler finish the last calculations before returning to its IDLE state, waiting for the next startofpacket on the next video data. The ready signal is low in this period preventing the scaler from missing a startofpacket signal.

4.6.2 Reverse Mapping

The reverse mapping function was implemented using the optimized equations from Equation 3.5. To be able to calculate these equations on a DSP, they were split into two as seen in Listing 2.

```
1  -- Fixed point DSP multiplication of variable part of dx/dy calculation
2  dx_1     <= x_count_ufx_reg * scaling_ratio_reg;
3  dy_1     <= y_count_ufx_reg * scaling_ratio_reg;
4  dx_1_reg <= dx_1;
5  dy_1_reg <= dy_1;
6
7  -- Constant part of dx/dy calculation
8  dxy_2      <= to_ufixed(0.5, 1, -2) * (1 - resize(scaling_ratio_reg, 12,
    ↪  -14));
9  dxy_2_reg  <= dxy_2;
10
11 -- Final dx/dy calculation
12 dx       <= dx_1_reg + dxy_2_reg;
13 dy       <= dy_1_reg + dxy_2_reg;
14 dx_reg   <= dx;
15 dy_reg   <= dy;
```

Listing 2: Reverse mapping calculation

First the $x' * c_1$ and $y' * c_1$ calculations from Equation 3.5 was calculated as seen in line 2 and 3 from Listing 2 using an 18x18 multiplication DSP on the Arria 10 FPGA. The 18x18 multiplication DSP supports two 18-bit unsigned multipliers with 37-bit output [12].

The x' and y' values were represented using an UQ12.0 fixed-point representation. This gave 12-bits for the integer part, meaning that an output resolution consisting of $2^{12} = 4096$ pixel in both width and height dimension is supported. There was no need for a fractional part since these numbers represent the pixels in the output video frame, which are of integer values.

For the c_1 value, seen as *scaling_ratio_reg* in Listing 2, an UQ3.12 fixed-point representation was used. Having a 3-bit integer part means that you could support downscaling as well, as this will give $c_1 > 1$. For the upscaling process the accuracy of c_1 is given

by the 12-bit fractional part, which gives a resolution of $2^{-12} \approx 0.000244$. This way the reverse mapping function will be very accurate while still being able to be calculated on the built-in DSP.

After the DSP multiplication, the c_2 value was added to the result, thus completing the reverse mapping calculation using Equation 3.5.

To keep the reverse mapping algorithm inside the 4-line framebuffer a check was implemented as seen in Listing 3.

```
1  -- Check if all rowns in line buffer is completed
2  if dy_reg >= C_LINE_BUFFERS then
3     -- Reset y_count for frambuffer addresses
4     y_count            <= 0;
5     y_count_ufx        <= to_ufixed(0, y_count_ufx);
6     y_count_ufx_reg    <= to_ufixed(0, y_count_ufx_reg);
7     -- Variable part of dx/dy is zero, use only constant part
8     dy                 <= resize(dxy_2_reg, dy'high, dy'low);
9     dy_reg             <= resize(dxy_2_reg, dy'high, dy'low);
10    dy_int             <= 0;
11 else
12    dy_int <= to_integer(dy_reg);
13 end if;
```

Listing 3: Keep dy within the framebuffer

This way when the dy-calculation stated that line number 5 in the framebuffer was up next, the dy-value was instead reset to start from line 0 thereby looping the four lines in the framebuffer until the frame was completed.

4.6.3 Nearest-Neighbor Interpolation

After calculating a pixels relative position in the input frame using the reverse mapping function, the framebuffer address of that pixel were calculated. This was done as seen in Listing 4

```
1  -- Use floor from my_fixed_pkg to get dx/dy to integer for fb_rd_addr
2  dx_int       <= to_integer(dx_reg);
3  dx_int_reg   <= dx_int;
4  dy_int_reg   <= dy_int;
5
6  -- Find nearest neighbor address for framebuffer
7  fb_rd_addr_i <= g_rx_video_width*dy_int_reg + dx_int_reg;
```

Listing 4: Nearest-neighbor interpolation

Here the floor function was performed by doing truncation. This way the digits right of the decimal point was limited, which is a very quick operation to do on an FPGA.

4.6.4 Bilinear Interpolation

Bilinear interpolation uses the same method for reverse mapping calculation as nearest-neighbor interpolation. However, bilinear interpolation requires a 2×2 neighborhood of pixels to calculate the new pixel value. The position of these pixels in the input frame needed therefore to be calculated, and this was done as seen in Listing 5.

```
1  if dx_reg < 1 then
2     x1_int <= 1;
3     x2_int <= 2;
4     dx_reg_1 <= to_ufixed(1, dx_reg);
5  elsif dx_reg > g_rx_video_width then
6     x1_int <= g_rx_video_width - 1;
7     x2_int <= g_rx_video_width;
8     dx_reg_1 <= to_ufixed(g_rx_video_width, dx_reg);
9  else
10    x1_int <= to_integer(dx_reg);
11    x2_int <= to_integer(dx_reg) + 1;
12    dx_reg_1 <= dx_reg;
13 end if;
14
15 if dy_reg < 1 then
16    dy_int   <= 1;
17    y1_int   <= 1;
18    y2_int   <= 2;
19    dy_reg_1 <= to_ufixed(1, dy_reg);
20 elsif dy_reg >= C_LINE_BUFFERS+1 then
21    -- Start from beginning of framebuffer when both lines have been
       ↪  completed
22    y_count           <= 1;
23    y_count_ufx       <= to_ufixed(1, y_count_ufx);
24    y_count_ufx_reg   <= to_ufixed(1, y_count_ufx_reg);
25    dy                <= resize(scaling_ratio_reg + dxy_2_reg, dy);
26    dy_reg            <= resize(scaling_ratio_reg + dxy_2_reg, dy);
27    dy_int            <= 1;
28    y1_int            <= 1;
29    y2_int            <= 2;
30    dy_reg_1          <= to_ufixed(1, dy_reg);
31 elsif dy_reg >= C_LINE_BUFFERS then
32    -- Special case when one line has completed but not the other one
33    dy_int   <= C_LINE_BUFFERS;
34    y1_int   <= C_LINE_BUFFERS;
35    y2_int   <= 1;
36    dy_reg_1 <= dy_reg;
37 else
38    dy_int   <= to_integer(dy_reg);
39    y1_int   <= to_integer(dy_reg);
40    y2_int   <= to_integer(dy_reg) + 1;
41    dy_reg_1 <= dy_reg;
42 end if;
```

Listing 5: Bilinear interpolation pixel position calculation

Here the pixels are being calculated as the 2×2 neighborhood around the dx- and dy-

position, while still making sure they are within the input frame. These pixel positions are then used to calculated the framebuffer address from Equation 3.10 and 3.11 as seen in Listing 6.

```
fb_rd_addr_a_i <= ((y1_int-1)*g_rx_video_width) + (x1_int - 1);
fb_rd_addr_b_i <= ((y1_int-1)*g_rx_video_width) + (x2_int - 1);
fb_rd_addr_c_i <= ((y2_int-1)*g_rx_video_width) + (x1_int - 1);
fb_rd_addr_d_i <= ((y2_int-1)*g_rx_video_width) + (x2_int - 1);
```

Listing 6: Bilinear interpolation framebuffer address calculation

To calculate the new pixel value, we use Equations 2.7, 2.8 and 2.9. However, these equations are not very well optimized for FPGA, so they were split up and parallelized. The first thing that was done was to calculate the weighting coefficient values. E.g. the weighting coefficient for pixel $f(x_1, y_1)$ would be $\frac{x_2 - x'}{x_2 - x_1}$. This was done as seen in Listing 7 where they are called delta values.

```
delta_x1 <= resize(dx_reg - x1_int, delta_x1);
delta_x2 <= resize(x2_int - dx_reg, delta_x2);
delta_y1 <= resize(dy_reg - y1_int, delta_y1);
delta_y2 <= resize(y2_int - dy_reg, delta_y2);
```

Listing 7: Bilinear interpolation weighting coefficient calculation

UQ1.16 fixed-point representation was used for these values. The 1-bit integer part were used for the case when the delta value is 1, that is when the reverse mapping maps hits the pixel position exactly on the original pixel from the input frame. A 16-bit fractional part ensures a high degree of accuracy with a resolution of $2^{-16} \approx 0.000015$, while still being able to fit on an 18x18 multiplication DSP. These delta values were then used to calculate the pixel values in a pipelined fashion as seen in Listing 8.

```
-- Calculate pixel values
A_y1_a <= delta_x2_reg * pix1_data_ufx_reg(7 downto 0);
A_y1_b <= delta_x1_reg * pix2_data_ufx_reg(7 downto 0);
A_y2_a <= delta_x2_reg * pix3_data_ufx_reg(7 downto 0);
A_y2_b <= delta_x1_reg * pix4_data_ufx_reg(7 downto 0);

A_y1 <= resize(A_y1_a_reg + A_y1_b_reg, A_y1);
A_y2 <= resize(A_y2_a_reg + A_y2_b_reg, A_y2);

A_1 <= resize(delta_y2_reg_4*A_y1_reg, A_1);
A_2 <= resize(delta_y1_reg_4*A_y2_reg, A_2);

-- Final calculation of new pixel value
A  <= resize(A_1_reg + A_2_reg, A);
```

Listing 8: Bilinear interpolation pixel value calculation

In Listing 8 the code for intermediate register assignment is not shown, but registers were used to achieve proper performance on the built-in fixed-point DSPs on the FPGA. Also

these calculation were performed for all three colour components, but only one is shown in the listing. The final step were to round the final pixel value to an 8-bit integer value, which is performed by doing truncation.

4.7 FIFOs

FIFOs were originally designed on the input to handle the receiving of data during a ready low state from the state machine seen in Figure 4.6. This works since the scaler is meant to operate at a higher frequency than the input data stream, allowing to empty the FIFO faster than it is being filled. By constantly filling the input FIFO at a steady rate, and then empty it at a high rate after each framebuffer line completion, a proper flow of data to the scaler is ensured while not having to stop the input flow of data. This does require dual-clocked FIFOs, which unfortunately where not implemented in this project due to time limits.

A single-clock generic FIFO was designed and implemented, which can be seen in Appendix A.1, but there was no time to expand this design to a dual-clocked FIFO.

Chapter 5

Verification IP Implementation

5.1 UVVM

UVVM (Universal VHDL Verification Methodology) is an open-source methodology and library used to improve testing and verification of VHDL modules. It is published by Bitvis on GitHub under the MIT licence [13], and it allows for making structured testbenches with the reuse of verification components (VCs).

The idea behind UVVM is reusability of previous implemented verification components for different testbenches. Using these verification components, together with the utilities provided, a lot of the groundwork for making testbanches has already been made. New functionality can then be added by extending the verification component instead of re-designing the testbench [2].

UVVM is made up of three main components: the utility library, the bus functional model, and the VHDL verification component. The user has the choice to only use a few of these components, or to use them all together to construct the testbench. This makes UVVM very flexible, as it can be used as a complete toolset for designing a testbench, or only as an additional utility to previously existing testbenches.

5.1.1 Utility Library

The utility library of UVVM consists of some fundamental components for verification. It includes support for logging/reporting, verbosity control, alert handling and predefined value and event checkers. This allows the user to automatically receive reports from the VHDL simulator when a violation or error occurs in the design.

Large design containing many modules and interfaces can be hard to verify, especially if the user relies on waveforms and manual checking of violations. This substantially increases the chance for an error or violation to slip past the user, and this can lead to

a faulty design. By having automated test reports presented to the user, the chance of catching an error is increased drastically, and thus the utility library of UVVM provides the foundation of creating an automated testbench and verification method.

5.1.2 BFM (Bus Functional Model)

A BFM (Bus Functional Model) is a non-synthesizable software model of a component with a set of tasks used to apply stimulus to a DUT (Device Under Test). It defines an interface similar to the underlying module of the DUT, which can be used to connect to the DUT and apply the stimuli in a similar fashion as the design would have received in a complete working system. This way the BFM can be used to simulate the connection between two modules or systems, and the data exchange between these in an operational system.

Sending test data to a DUT from a testbench is simplified with the use of a BFM as it works under the principle ”implement once, use many”. A simple example of writing data to a FIFO using a BFM compared to a classical testbench approach can be seen in Table 5.1.

	Normal testbench	UVVM BFM
1	write_en <= ’1’;	
2	data <= x”F6A4”;	write_fifo(x”F6A4”);
3	wait until rising_edge(clk);	
4	write_en <= ’0’;	

Table 5.1: Writing data to FIFO from testbench using BFM

In this example the user would have implemented a BFM that handled the assertion of the write-enable signal, sending of data, and then setting the write-enable signal low after completion. This way you could reuse the BFM whenever you needed to write some data to the FIFO.

5.1.3 VVC (VHDL Verification Component)

Using BFMs and invoking these calls sequential in the testbench might not help you to catch all the edge cases when it comes to verifying your design. This is where a VVC (VHDL Verification Component) comes in to play. You can send commands and test-patterns to a VVC, and these commands will be queued inside the VVC and later executed towards the DUT. UVVM support multiple threads and several VVC in parallel, making it easier to catch weird edge cases involving several interfaces being targeted at the same time.

A VVC is made up of three main stages: Interpreter, Command Queue, and Executor. These stages can be seen in Figure 5.1.

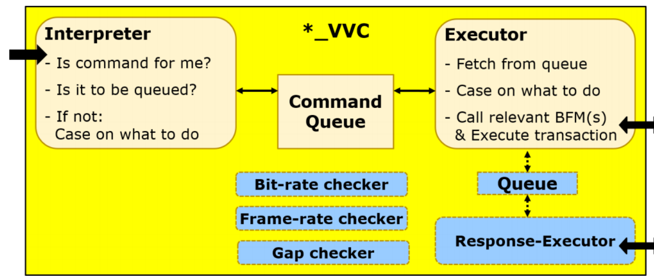


Figure 5.1: Structured VVC architecture [2]

The Interpreter receives a command from the sequencer, checks if the command should be queued, and puts the command into the Command Queue if this is the case. The Executor then fetches a command from the Command Queue, and calls the relevant BFM to execute this command. This way queuing and execution of a series of commands is handled by the VVC, and a large number of test-cases can be handled with a relative ease in the testbench.

5.2 Avalon-ST VIP

A VIP (Verification IP) was developed for this project with support for the Avalon-ST Video interface, as this did not exist amongst the VIPs included with UVVM, or within the open-source UVVM community. This implementation will be uploaded with a MIT licence to the open-source UVVM community for others to use and further develop if they so desire. The source code for the BFM implementation can be seen in Appendix C, and the complete source code can be found on the GitHub repository as stated in Appendix C.

5.2.1 Avalon-ST BFM

The BFM was implemented to support transmission of serial data using the Avalon-ST Video interface as specified in Intel Video and Image Processing Suite User Guide [6], with the possibility of further extension to support the full Avalon-ST interface specified in [5]. It consists of three main functions: avalon_st_send, avalon_st_receieve, and avalon_st_expect.

Avalon_st_send is the function used for transmitting a packet consisting of data-symbols in a serial manner. It uses a ready latency of 1 clock cycle, as described in Chapter 4.3.1, with startofpacket and endofpacket signals to mark the first and last symbol of the packet being sent.

The avalon_st_send function works by generating a slv-array, which is a sub-type of a std_logic_vector array with an unconstrained width, that can be dynamically increased in terms of depth to support the number of data-symbols in the given package. This way the slv-array can be filled with data from the testbench without having to be re-compiled to

support the new symbol widths of the data packet sent. The BFM then loops through this slv-array, sends one entry at a time, and assert the startofpacket and endofpacket signals on the first and last slv-array respectively together with the valid signal when it receives a ready signal. If the ready signal goes low, the BFM sends one more data-symbol according to the ready latency of 1, before pausing the transmission until the ready signal is asserted again.

The maximum width of the data-symbols supported is limited at compile time to the maximum data-width supported by the Avalon-ST interface, which is 4096 bits [5]. The maximum depth of the slv-array is set in the VVC command package, and this is limited by the amount of RAM present at the computer running the simulation, as the slv-array is mapped to memory at compile time based on these values, and not the amount of data written to the slv-array from the testbench.

To receive output data from a module, the avalon_st_receive function is used. This function works in similar manners to the avalon_st_send function in that it uses a slv-array to hold the data. Avalon_st_receive asserts its ready signal and waits for a startofpacket signal together with a valid signal. When these two signals are received, avalon_st_receive fills a slv-array with the incoming data until the endofpacket signal is received. After the transmission is done, the slv-array is returned as an array to the testbench.

Avalon_st_expect further builds on the avalon_st_receive function in that it can have a slv-array as an input, and use this array to do data comparison. The avalon_st_expect function uses the avalon_st_receive function to receive data and store this in a slv-array, and then it compares this received slv-array with the one provided as an input from the testbench. If there are some data entries in these two arrays that do not match, an error is raised. This way the avalon_st_expect function can be used to validate the data received against known data values.

5.2.2 Avalon-ST VVC

To be able to send and receive multiple entries of test data in an easy manner, two VVCs were implemented for the Avalon-ST VIP, the Avalon ST Source VVC and the Avalon ST Sink VVC. By splitting the sending and receiving of data into two VVCs, a greater possibility of detecting edge cases from concurrent send and receive is achieved. This way you could verify designs that is built around having a single input while producing several outputs, i.e. a video scaler with several outputs having different resolutions. The Source VVC works as the sender module in this case, calling the avalon_st_send function from the BFM, while the Sink VVC acts as a receiver utilizing avalon_st_receive and avalon_st_expect.

Support for random ready signaling is implemented in these VVCs. That is asserting and de-asserting the ready signal at random times. This is done by utilizing the random function from UVVM to generate a random number between 1 and 100, and then see if this is bellow some value specified in the configuration of the VVC. If the VVC is set up with a value of 50, the ready signal will only be asserted when the random number from UVVM is between 1 and 50, effectively asserting the ready signal 50% of the total time

at random times. This can be used to better detect edge cases when you have a receiver module that asserts its ready signal at odd times.

5.2.3 Memory Concerns

As stated the BFM was implemented using a dynamically slv-array. The reason for this choice was to be able to send test data of different symbol-width, and of different number of symbols in a packet, without having to re-compile the Avalon-ST VIP for each test case. However, this could potentially take up a lot of system RAM, as the entire slv-array is mapped to memory at compile time even though only a part of this array is used by the testbench. Some experimental results using a slv-array of width 4096-bits with a depth of 65 536 entries gave a system RAM usage around 4 GB when compiled and simulated using ModelSim DE-64 10.7c. Thus a problem may arise when large data-sets are being used with this VIP.

A possible solution to this could be to add file I/O functionality to the Avalon-ST VIP, where the slv-array could be replaced with reading data values directly from a file. However, this was not done in this project due to time limitations.

Chapter 6

Testing and Verification Strategy

6.1 Verifying VHDL Modules

The first step in verifying the design will be to test the different sub-modules by using UVVM and the Avalon-ST VIP implemented. This way a randomized testing strategy can be used where test are running a random number of times, and valid/ready flow control is tested with random test patterns.

The controller sub-module will be tested using this method of testing. Unfortunately, due to time limits, the scaler was not completed in such a way that the controller could start and stop the scaler according to the intended design. Because of this the Avalon-ST VIP implemented could not be used to test the scaler itself, and thus a more classical testbench approach was used to test the scaler together with test images.

The other sub-modules such as the FIFO and RAM designs was tested using classical testbench approaches with manual validation of their correctness. The result from this testing will not be presented in the results section, as these modules are considered to be simple modules. The testbench code used for these modules is provided in Appendix B, together with all the other testbenches, for the interested reader.

6.2 Scaler Verification and Image Quality

To test the scaler design a testbench approach with file I/O was used. This way test data could be sent to the scaler module using known data from a local file, and the output from the scaler could be stored in a new file for later comparison with the original material. MATLAB was used for this verification and comparison, as this was the platform that had been used in the project thesis to perform quality comparison between the different scaling algorithms [4].

This testing was performed by using a series of test images stored at 1920x1080 resolution in the 8-bit PNG file format, downscaling these images to 360p, 540p and 720p resolution using MATLABs built-in bicubic interpolation algorithm, and then use these down-scaled images to scale back up to 1080p resolution. These new up-scaled versions was then compared against the original 1080p version of the test image together with up-scaled versions created by using MATLABs built-in image processing toolbox. MATLABs image processing toolbox would then act as a reference, as one can expect these implementations of the nearest-neighbor and bilinear interpolation algorithms to be of good quality. To have a fair comparison between MATLAB and VHDL, the anti-aliasing filter of MATLABs scaler was turned off so that it would only use the raw interpolation algorithm.

6.2.1 Matlab binary conversion

To be able to use the test images with the VHDL testbench they had to be in a binary format. As this scaler design uses the Avalon-ST Video interface which sends pixel data in a serial fashion, a natural way was to do this in the testbench as well. A MATLAB function was therefore needed to convert the PNG images to a serial binary data file. This was done with the *img2bin* function presented in Appendix D.1. Using 8-bit pixel values this code generates a 24-bit wide binary file where one pixel is represented on each line in the file. The testbench then reads one line of the binary file per clock cycle, and send this data to the scaler as seen in the testbench code in Appendix B.4.

The output from the scaler is binary 24-bit data, same as the input, and this data is stored to a binary file similar to the input file. A function converting this binary file back to a PNG image was made, and this function can be seen in Appendix D.2. The image created by this function was then used as comparison with the original image against MATLABs image processing toolbox.

6.2.2 Objective image quality models

Performance of the VHDL implementation in terms of image quality was determined by using objective image quality models. The human eye can quite easily distinguish good image quality from bad, but when two samples are very close to each other, it is not a very good tool to determine which one is the best. To be able to differentiate two samples from each other, mathematical models have been developed to approximate the results from subjective quality assessments. By having an objective method to measure image quality, the process of comparing two samples can be more accurate.

Full reference (FR) methods aims to give a quality metric by comparing the original video material to the received one. This is typically done by comparing each pixel in the received signal against the corresponding pixel in the original signal. This makes FR models usually the most accurate for determining objective quality. Examples of widely used FR models to determine image quality are Peak Signal-to-Noise Ratio (PSNR), Mean-Squared Error (MSE) and Structural Similarity (SSIM).

Peak Signal-to-Noise Ratio (PSNR) is the most widely used objective image quality metric. It gives the ratio between the maximum possible power of a signal and the power of corrupting noise. PSNR is built upon the Mean-Squared Error (MSE), which is the average of the squares of the errors.

While PSNR and MSE produces a quality metric based upon an estimate of *absolute errors*, Structural Similarity (SSIM) tries to estimate the quality of an image as *perceived change in structural information*. An image can be perceived by the human eye as having good quality if the structure of the objects in the image is intact, even though there is a loss of information at a pixel-by-pixel level. Thus SSIM is designed to improve on traditional methods for quality estimation by ranking the quality of the image as perceived by the human eye.

6.2.3 Test Images

The test images used for the objective image quality comparison were chosen to be of both animated and natural content. This way a lot of the content normally displayed would be represented. Because of time limitations and due to the fact that the scaler design was not fully completed and connected to the controller module, a single test image was used at a time.

A more preferred way of testing would be to have many images from the same content, or even an entire video clip consisting of many frames. This way the result would be more accurate than when only having a single image from a given content, as this one image could be of an ”unlucky” nature and thus give the scaler a poorer performance metric that it would have gotten otherwise. However, the same exact images was used for both the VHDL implementation and the MATLAB, so they should be comparable to each other.

Animated content was represented using screengrabs from the two animated movies Lion King and Toy Story. These images can be seen in Figure 6.1.

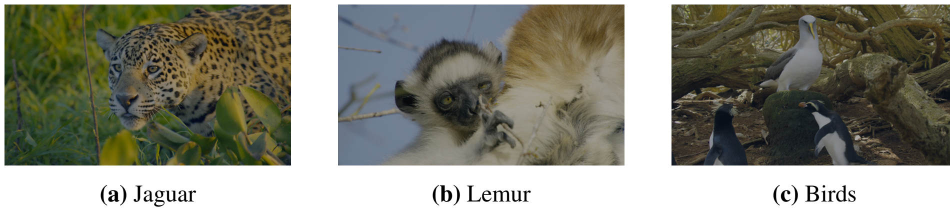
(a) Lion King [14]

(b) Toy Story [15]

Figure 6.1: Animated content used for testing

These images was captured from the Blu-Ray release using the FFmpeg software library [16] to do a screengrab.

Natural content was represented using screengrabs from BBCs nature documentary Planet Earth II. These images can be seen in Figure 6.2.



(a) Jaguar **(b)** Lemur **(c)** Birds

Figure 6.2: Natural content used for testing. Taken from Planet Earth II [3]

These images was also captured by using the FFmpeg software library. However, they were captured from the 4K UHD Blu-Ray release, and therefore they were first downscaled to 1080p using MATLABs built-in bicubic interpolation algorithm to be of the same resolution as the other test images.

6.3 Synthesis test

The synthesis test is done to ensure that the design runs on an FPGA at the targeted specifications for frequency, area, resource usage and DSP usage. To simulate a ”worst case” most heavy workload, the synthesis test is done with upscaling from 1080p to 2160p. This ensures that the design will run on an FPGA under these heavy workload conditions. The results from the synthesis test are taken from the ”balanced” compilation and packing approach in Quartus 19.1, and the slow performance model, to ensure that it operates under ”worst case” conditions.

The synthesis test does not include extra designs to intentionally pack the FPGA full of several designs, so a target frequency of 300 MHz is set in this case, with an expected operational frequency of 250 MHz. This allows for some headroom if there are several other designs running on the same FPGA which require extra packing of the design.

In the synthesis test a dummy source and a dummy sink is attached to the design. This prevents the compiler from optimizing away parts of the design that would otherwise have had open connections. This ensures that the interface and all the signals in the design stays intact during the compilation and optimization process.

Chapter 7

Results and Discussion

7.1 Verification of Sub-Modules

The verification of sub-modules were done using UVVM to test the scaler wrapper and the scaler controller. Functional verification of the FSM were performed by using avalon_send() and avalon_expect() to ensure that the FSM in the controller goes to the correct state, that new control packets are being produced when the controller is in the ctrl_pkg_gen state, that unsupported packages are passed through, and that video data is passed to the scaler in the video_data state.

Unfortunately, due to time limits, the complete scaler implementation with the top level wrapper were not completed, so the scaler itself were not tested and verified with UVVM. The scaler implementation is thus only tested with visual functional verification. The testbenches used for this testing can be ween in Appendix B.

7.1.1 Controller

In the test output shown in Listing 9 the avalon_st_send() function implemented in the Avalon-ST VIP for UVVM was used to send data to the controller module through the scaler wrapper, and verify the packet decoder function of the controller. This data consisted of the packet identifier of a control packet, as well as the input resolution of the video given in the Avalon-ST Video format. The receiver was the avalon_st_expect() function, and this function did checking to ensure that the data from the newly generated control packet from the controller was correct.

```
# UVVM: 100.0 ns  Sending control packet
# UVVM: 100.0 ns  check_value() => OK, for boolean true.
 ↪  'avalon_st_send(AVALON_ST_VVC,1, 2 data entries)data_array length
 ↪  must be > 0'
# UVVM: 100.0 ns  ->avalon_st_send(AVALON_ST_VVC,1, 2 data entries):
 ↪  'Sending v_data_array'. [9]
# UVVM: 100.0 ns  ACK received.   [9]
# UVVM: 100.0 ns  ->avalon_st_expect(): 'Checking data'. [10]
# UVVM: 100.0 ns  ACK received.   [10]
# UVVM: 100.0 ns  ->await_completion(AVALON_ST_VVC,1,rx, 23040000 ns): .
 ↪  [11]
# UVVM: 102.0 ns  avalon_st_send(80 bits) => 'Sending v_data_array'  [9]
# UVVM: 102.0 ns  avalon_st_receive(80 bits) => 'Checking data'  [10]
# UVVM: 125.0 ns  avalon_st_send(80 bits) => Sent 2 data entries
# UVVM: 135.0 ns  avalon_st_receive(80 bits) => Received 2 data entries
# UVVM: 135.0 ns  avalon_st_expect(80 data bits, 1 empty bits)=> OK,
 ↪  received 2 data entries. 'Checking data'  [10]
# UVVM: 135.0 ns  ACK received.   [11]
# UVVM: >> Simulation SUCCESS: No mismatch between counted and expected
 ↪  serious alerts
```

Listing 9: UVVM output from controller packet decoder test

As seen from the listing two data entries were sent, the packet identifier and the control packet containing the resolution information. The avalon_st_recieve() function received these two packets, and the avalon_st_expect() did check the result against the expected result with no errors. Thus the conclusion from this test is that the controller managed to interpret an incoming control packet, and generate a new one based on the output resolution of the video.

This test was also repeated using a random ready-pattern, that is asserting the ready signal high at random intervals to simulate delays in the receive process, and by using video data packets that were bypassed the scaler module. This was done to see if the controller would handle video packets in a correct way. The results from this testing was also successful and the controller did not lose any packets, which can be concluded with that the valid/ready flow of the controller works as expected.

7.2 Scaler Verification and Image Quality

Functional verification of the implemented scaler was done by converting the output data from the scaler to a PNG image, and compare this with the original image using a MATLAB script with PSNR, MSE and SSIM tests. The output produced by the scaler was also verified by doing a visual functional verification where the output image was compared to the original image in search for unwanted errors and artifacts in the image. The complete test results from this testing can be seen in Appendix E.

7.2.1 Nearest-Neighbor Functional Verification

Two samples from the output images of the scaler using nearest-neighbor interpolation upscaling from 720p to 1080p can be seen in Figure 7.1.

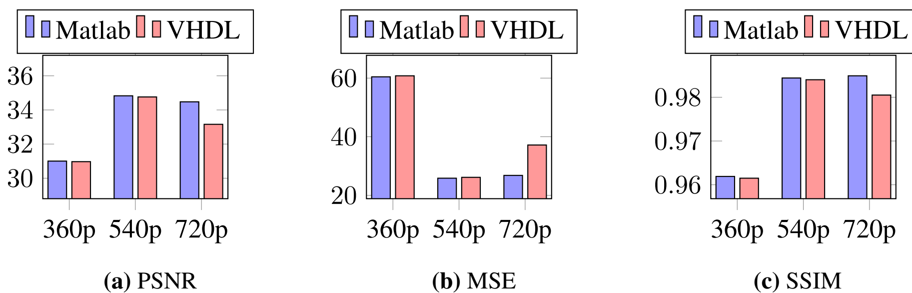
(a) Lion King **(b)** Planet Earth II Jaguar

Figure 7.1: Output image from nearest-neighbor upscaling from 720p to 1080p

The two samples from Figure 7.1 were produced using a 4-line framebuffer and the implemented nearest-neighbor scaling algorithm in VHDL. Looking at these images we can see no visual errors or artifacts when comparing with the source material seen in Figure 6.1a and Figure 6.2a. A conclusion to this visual inspection would be that the nearest-neighbor implementation is producing an output as expected.

7.2.2 Nearest-Neighbor Image Quality

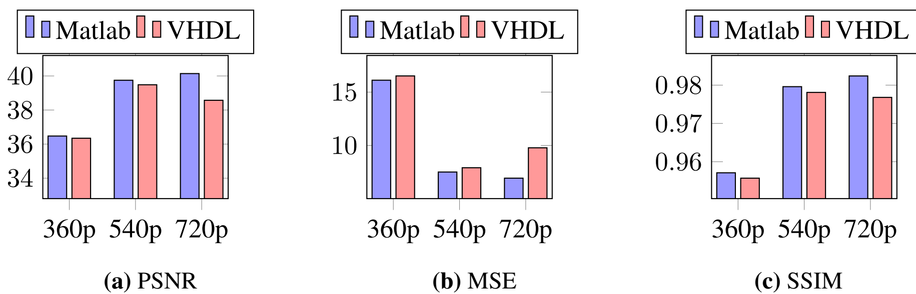
Nearest-neighbor upscaling performance can be seen in Figure 7.2 and Figure 7.3, which contains data from upscaling animated and natural content to 1080p respectively.



(a) PSNR **(b)** MSE **(c)** SSIM

Figure 7.2: Nearest neighbor upscaling to 1080p using animated content

We can see from Figure 7.2 that the VHDL implementation performs identical to the MATLAB implementation when using 360p and 540p source material. However, when looking at the 720p source material, we see that the VHDL implementation falls behind in all three tests.



(a) PSNR **(b)** MSE **(c)** SSIM

Figure 7.3: Nearest neighbor upscaling to 1080p using natural content

Looking at Figure 7.3 we see a similar pattern using natural content as we saw with the animated content. The 360p and 540p source material performs about equal as the MATLAB algorithm, while the 720p source material falls behind in all three tests.

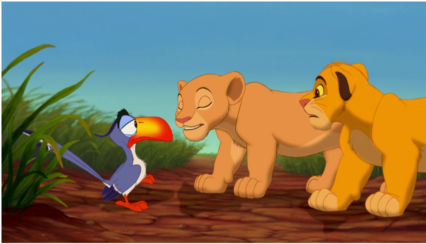
A possible explanation for this is that 720p to 1080p scaling has a scale ratio of $2/3 = 0.666666...$, which cannot be accurate represented using fixed point numbers in VHDL. This will give values either too low or too high on certain occasions, resulting in a rounding error when calculating the framebuffer address.

On a $2/3$ scaling ratio, every 3rd pixel would have an value with zero decimal points. Looking at the 3rd pixel this would have the value 2 as framebuffer address in MATLAB, whereas for VHDL using 12-bit decimal notation for scaling ratio calculation, this corresponds to $2/3 = 0.666504$, which gives the 3rd pixel a value of $0.666504 * 3 = 1.999512$. Since this design used a floor function to determine the framebuffer address this would give the value 1 as the framebuffer address, and thus the output pixel would be wrong compared to the MATLAB scaler.

So why does this not happen on 360p, which uses a scaling ratio of $1/3$? Looking at the scaling ratio calculation using 12-bit decimal points in VHDL we see a result of $1/3 = 0.333252$. This would also lead to a rounding error with the floor function on every 3rd pixel. However, the 360p source material is a much more ”blurred-out” image compared to the 720p. There is a lot less detail in that image, and this might be why we do not see a more dramatic effect with 360p source material. There is simply not very much detail that are being lost even though we get rounding errors, and this might be why the VHDL implementation perform equal to the MATLAB implementation.

7.2.3 Bilinear Functional Verification

Two samples of the output images produced using a full sized framebuffer and bilinear interpolation with upscaling from 720p to 1080p can be seen in Figure 7.4.



(a) Lion King

(b) Planet Earth II Jaguar

Figure 7.4: Output image from bilinear upscaling from 720p to 1080p

The samples from Figure 7.4 were produced using a framebuffer the same size as the input image. As seen from the two samples, there are no visual artifact or distortion in the images, which from a visual functional verification standpoint can be concluded with a proper working bilinear interpolation implementation. However, using a framebuffer the size of the input image is not a good solution for a system that is designed to run on an FPGA, as this will take up too much memory on the FPGA.

Given an input image of 1080p which is upscaled to 2160p, this would require an framebuffer of size 1920px $*$ 1080px $*$ 24-bit $*$ 4-fb $=$ 199 Mbit. Given that the Intel Arria 10 GX 1150 has 54 Mbit of M20K memory built-in [11], this would simply not work. Therefore a solution is to buffer only a few lines in the framebuffer as was done with the nearest-neighbor implementation.

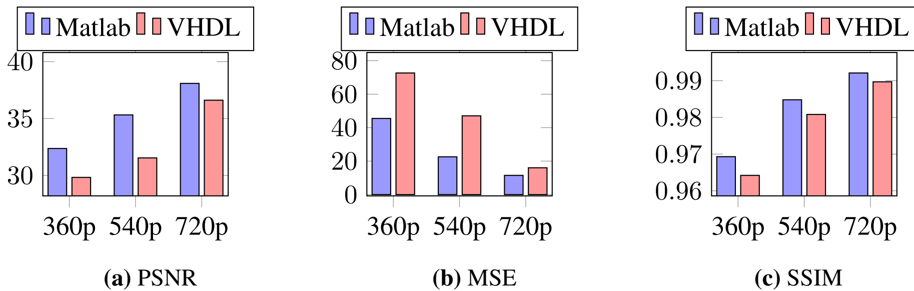
Using 4 lines in the framebuffer for the bilinear interpolation, a bug arises at the right-hand side of the output image. This can be seen in Figure 7.5.

Figure 7.5: Bilinear bug using 4-line framebuffer

The last two pixels on the right hand side of the image consist of data belonging to the left hand side of the image. Given that the bilinear implementation uses the same framebuffer as nearest neighbor, it is most likely not the framebuffer that is the problem. This was also confirmed by manually checking each read address sent to the framebuffer. It is believed that this bug arises due to dx/dy calculations after the reset of dy position to 1 when reaching the last line in the framebuffer. A big effort was put into locating this bug, but due to time limits this search had to be abandoned. Because of this bug, the output images used in the image quality comparisons were generated using the full size framebuffer.

7.2.4 Bilinear Image Quality

Bilinear upscaling performance using animated and natural content can be seen in Figure 7.6 and Figure 7.7 respectively.

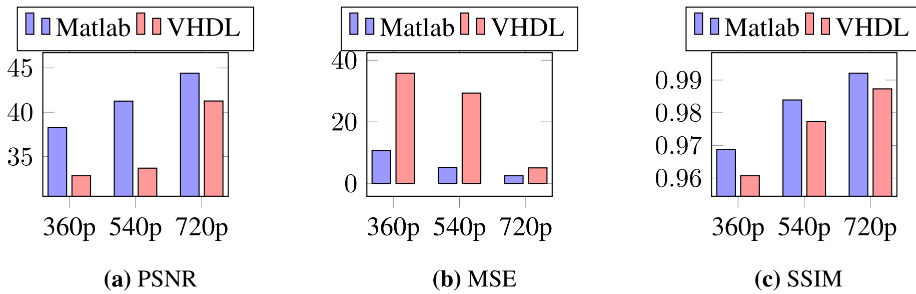


(a) PSNR **(b)** MSE **(c)** SSIM

Figure 7.6: Bilinear upscaling to 1080p using animated content

Looking at the results from bilinear upscaling to 1080p using animated content in Figure 7.6, we see a quite different results than what we saw with nearest-neighbor interpolation. All the tests shows that MATLABs built-in scaler outperforms the VHDL scaler in this design. This is true for both 360p, 540p and 720p source material, but the difference is much greater for the 360p and 540p material. This is opposite to what we had with the nearest-neighbor interpolation.

Using natural content as source material we get the result as seen in Figure 7.7.



(a) PSNR

(b) MSE

(c) SSIM

Figure 7.7: Bilinear upscaling to 1080p using natural content

From Figure 7.7 we see the same pattern as we saw when using animated content as the source material. MATLABs built-in interpolation algorithm outperforms the VHDL-based one in this design, with the worst results seen using 360p and 540p source material.

An explanation for the poor performance in the VHDL implementation could be the fixed-point representation used in this design. As we saw with nearest-neighbor using 720p content, fixed-point numbers are unable to precisely represent a fraction. In bilinear interpolation this effect is amplified by not only calculating dx/dy using fixed-point numbers, but by also by calculating the weighting coefficients for the kernel and the pixel values using fixed-point numbers. This way we get several levels of errors that passes through the design and becomes amplified with each new error. This results in an incorrect pixel value compared to calculation with MATLABs superior precision. When this happens for every pixel in the image, the end result compared to MATLAB would yield a poor outcome.

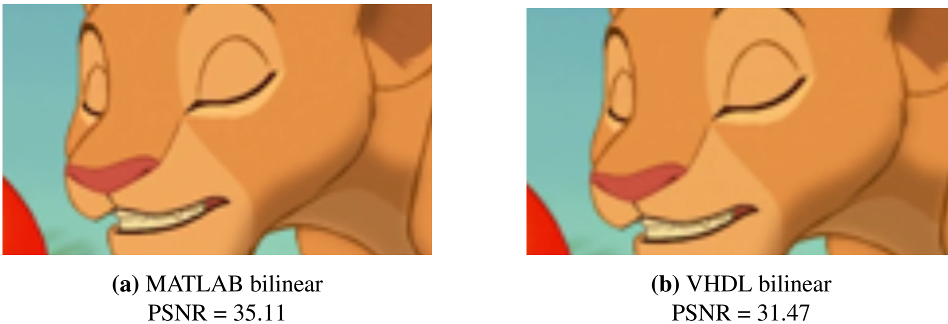
The strange thing though is that the upscaling performance using 720p source material is so much closer to the MATLAB results than what is the case using 360p and 540p source material. One possible explanation for this could be that the 720p source material simply has that much more data for the scaler to work with. Thus the rounding error could have a less severe impact on this, as it is out-weighted by the fact that there is more information in the image from the beginning.

An interesting note looking at Figure 7.6 and Figure 7.7 is that upscaling in VHDL using 540p source material is not much better than 360p. From the MATLAB results we see the expected increase in performance going from 360p to 540p and then to 720p, but for the VHDL implementation this is not the case, especially when looking at the PSNR results. We only see a small increase in performance from 360p to 540p, and then a huge performance increase going from 540p to 720p. This is a very strange result that is hard to explain.

7.2.5 Subjective image quality

Given the poor performance of the VHDL implementation using bilinear interpolation compared to MATLABs built-in scaler, the output images was used as a basis for a visual comparison between the two to see if the human eye could detect noticeable quality dif-

ferences between the two. The image from Lion King was used to represent the animated content, and it was the 360p source image that was chosen based on the big difference in PSNR between MATLAB and VHDL for this image. The result heavily zoomed in can be seen in Figure 7.8.



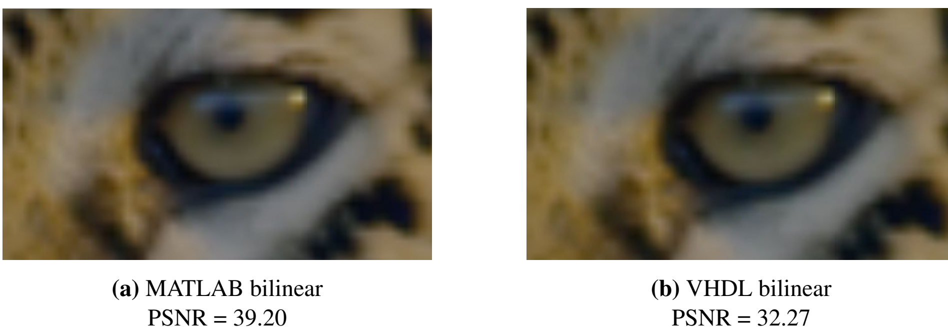
(a) MATLAB bilinear
PSNR = 35.11

(b) VHDL bilinear
PSNR = 31.47

Figure 7.8: MATLAB vs VHDL bilinear upscaling from 360p to 1080p using animated content

Taking a first glance at the two images side by side from Figure 7.8, we can see no obvious difference in image quality. There are no artifacts in the image or extra blurriness that the eye can see. Studying the two images on a computer monitor side by side might give a small hint at poorer colours in the VHDL image in Figure 7.8b. The MATLAB image might have a tint more red-ish colour in the lions fur while the VHDL image is a bit more ”washed out”, but you need to really study hard to see this difference.

The same was done with a sample image from the natural content. Here the Jaguar image from Planet Earth II was chosen, again because of its big difference in PSNR performance between the two images. The two images can be seen in Figure 7.9.



(a) MATLAB bilinear
PSNR = 39.20

(b) VHDL bilinear
PSNR = 32.27

Figure 7.9: MATLAB vs VHDL bilinear upscaling from 360p to 1080p using natural content

Looking at these two images side by side in Figure 7.9, we can see no obvious difference with no extra artifacts or blurriness in the VHDL image compared to the MATLAB image. In contrast to the animated content, no difference in colour can be seen either between these two images by studying them closely side by side on a computer monitor. To the human eye there is no difference between these two images.

The reason for the big performance between the MATLAB images and the VHDL images could be explained again with the fixed-point rounding errors. These errors contribute to pixels being in wrong places, and to colours having a bit off-values compared to implementations using infinite precision. The incorrect position of some pixels is something that the human eye cannot see, but a computer picks up on this quite easily. Incorrect colours can be seen in the animated content where large areas is of the same colour, but in natural content where variation in colour is much more rapid per pixel, it is almost impossible to notice for the human eye. Because of this we cannot say that the VHDL images would be perceived as having lower image quality by human viewers just because the objective quality measurements point to this.

7.3 Synthesis test

Synthesis tests were done to see how well the design would perform if it were synthesized to an actual FPGA. All test were performed using Intel Quartus 19.1, except the results seen in Table 7.2 where Intel Quartus 18.1 was used. The scaler design were initialized as an upscaling process from 1080p to 2160p using 8-bit pixel values. This way the synthesis test would be performed in one of the hardest scaling conditions in regards to framebuffer size and calculations needed.

The results are based on three metrics: Fmax, uTco and logic levels. Fmax is the maximum frequency the design can run at on the given FPGA device. The uTco metric is the ”time clock output” results, which is the time it takes between having rising edge on the clock, to having stable data on the output. Logic levels represent how many logic levels there are in the design. A good rule of thumb is to have no more that 5 logic levels in a design.

Results from the synthesis test performed on the different sub-modules of the scaler design can be seen in Table 7.1.

Sub module	Fmax	uTco	Logic levels
FIFO - M20K (24x1024-bit)	418.76 MHz	0.594 ns	5
Simple DP RAM - M20K (24x15360-bit)	365.10 MHz	0.591 ns	5
Multiport RAM - M20K (24x15360-bit)	347.83 MHz	0.591 ns	5
Scaler controller	388.05 MHz	0.250 ns	5

Table 7.1: Synthesis test of sub modules

As seen from Table 7.1 every sub-module performs exceptionally well. The performance goal for this design was to have no more that 5 logic levels, and to be able to achieve a Fmax of 300 MHz. This is achieved with good margins for all the different sub-modules, so they can be used in the scaler design without the concern of them slowing the design down.

Compilation and synthesis of the nearest-neighbor design using Intel Quartis 18.1 during the development process gave some strange results. The design were unable to properly

synthesize in such a way that DSPs were used for the fixed-point calculations. The synthesis of the design using Intel Quartus 18.1 gave the following results as seen in Table 7.2.

Scaling method (UQ-bits)	Notes	Fmax	Logic levels	DSPs
Nearest neighbor (UQ16.16)	no DSP	42.48 MHz	31	0
Nearest neighbor (UQ16.16)	no pipeline	121.00 MHz	5	4
Nearest neighbor (UQ16.16)	pipelined	144.76 MHz	6	3

Table 7.2: Synthesis test using Intel Quartus 18.1

The first test with the note ”no DSP” did not utilize any DSPs, and the entire design was compiled to a logic circuit. This gave the poor performance seen in the results. It is quite clear when looking at the Fmax and logic level results that this were synthesized as a large logic circuit. In the next results, the design was able to use DSP, which is seen by the increase in Fmax and the number of DSPs going from 0 to 4. However, the design was non-registered and had a poor pipeline performance, which holds back on the performance. Effort was made to pipeline the design more and to have proper register usage in the DSP. Splitting up the dx/dy calculation resulted in an increase in performance and lower DSP usage, as each calculation now became simplified, and more calculations could share a single DSP, and the final result of this gave Fmax of 144.76 MHz with 3 DSPs used.

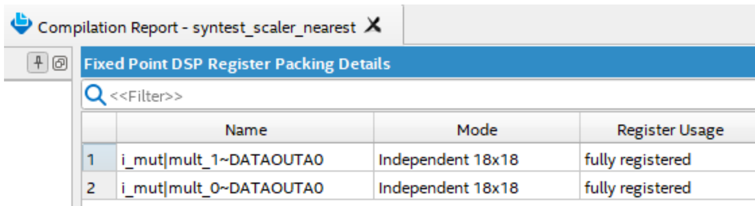
After many trials and errors the conclusion to the testing was that Quartus 18.1 did not recognize constants used in the calculations properly, and thus were unable to generate proper register-based DSP chains of the calculations. This ended with a switch to Quartus 19.1 to see if this gave a better results.

Quartus 19.1 were able to better recognize the constant values of the calculations, and thus a higher performance was achieved. By using Quartus 19.1 it also became more clear where the problem in the pipeline was, as it gave a better picture of the DSP chain. After further optimizations and splitting of equations, the pipeline became fully registered, and the results from this can be seen in the results in Table 7.3.

Scaling method (UQ-bits)	Fmax	uTco	Logic levels
Nearest neighbor (UQ16.16)	340.37 MHz	0.216 ns	4
Bilinear (UQ16.16)	322.79 MHz	0.216 ns	3

Table 7.3: Performance on Intel Arria 10 GX 1150

From table 7.3 we can see that nearest neighbor interpolation increased its Fmax from 144 MHz to 340 MHz. This was the results of using Quartus 19.1 that properly recognized the constant values of the calculations, thereby making the DSP pipeline fully registered as seen in Figure 7.10.


Compilation Report - syntest_scaler_nearest

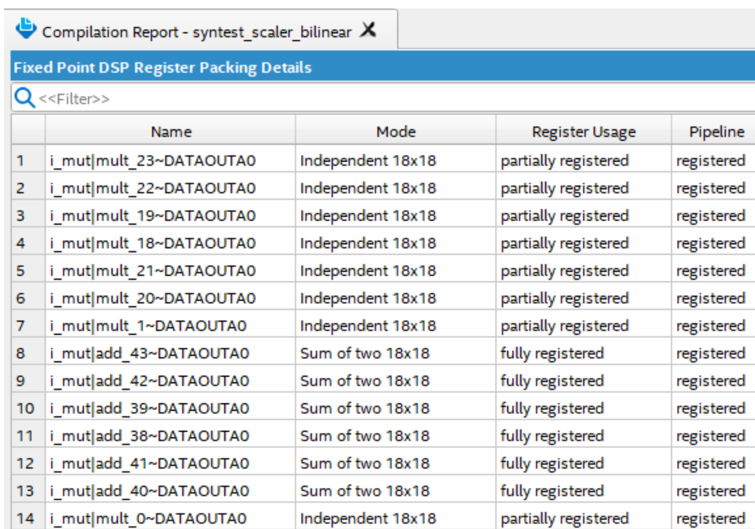
Fixed Point DSP Register Packing Details

	Name	Mode	Register Usage
1	i_mut|mult_1~DATAOUTA0	Independent 18x18	fully registered
2	i_mut|mult_0~DATAOUTA0	Independent 18x18	fully registered

Figure 7.10: Synthesis test nearest neighbor DSP pipeline

These final result made the nearest-neighbor interpolation meet the performance requirements of 300 MHz Fmax, and maximum 5 logic levels.

After making the nearest-neighbor calculation properly registered, it was quite easy to achieve the same performance result with bilinear interpolation. As seen from Table 7.3, bilinear interpolation achieves a Fmax of 322 MHz and uses 3 logic levels, which is well withing the performance requirements. However, the DSP chain of the bilinear calculation were not utilizing all of the recomended input and output registers as seen in Figure 7.11.


Compilation Report - syntest_scaler_bilinear

Fixed Point DSP Register Packing Details

	Name	Mode	Register Usage	Pipeline
1	i_mut|mult_23~DATAOUTA0	Independent 18x18	partially registered	registered
2	i_mut|mult_22~DATAOUTA0	Independent 18x18	partially registered	registered
3	i_mut|mult_19~DATAOUTA0	Independent 18x18	partially registered	registered
4	i_mut|mult_18~DATAOUTA0	Independent 18x18	partially registered	registered
5	i_mut|mult_21~DATAOUTA0	Independent 18x18	partially registered	registered
6	i_mut|mult_20~DATAOUTA0	Independent 18x18	partially registered	registered
7	i_mut|mult_1~DATAOUTA0	Independent 18x18	partially registered	registered
8	i_mut|add_43~DATAOUTA0	Sum of two 18x18	fully registered	registered
9	i_mut|add_42~DATAOUTA0	Sum of two 18x18	fully registered	registered
10	i_mut|add_39~DATAOUTA0	Sum of two 18x18	fully registered	registered
11	i_mut|add_38~DATAOUTA0	Sum of two 18x18	fully registered	registered
12	i_mut|add_41~DATAOUTA0	Sum of two 18x18	fully registered	registered
13	i_mut|add_40~DATAOUTA0	Sum of two 18x18	fully registered	registered
14	i_mut|mult_0~DATAOUTA0	Independent 18x18	partially registered	registered

Figure 7.11: Synthesis test bilinear DSP pipeline

As seen in Figure 7.11 some DSPs are only party registered. However all DSPs have registered pipeline, which is the factor affecting Fmax the most. More optimization of the pipeline with proper register usage on inputs and outputs could have given even better Fmax performance, but given that bilinear interpolation is a much more complicated task with heavier resource usage than nearest neighbor, one would not expect the bilinear performance to surpass nearest neighbor performance. Thus a bilinear Fmax performance of 94.8% of nearest neighbor is considered a very good result. Given the fact that bilinear interpolation with a 4-line framebuffer also did produce some artifacts on the right-hand side of the image, no more effort were put into optimizing this DSP chain any further.

The final part of the synthesis test was to see how high of a resource usage the different scaling methods used on the FPGA, and the results from this can be seen in Table 7.4.

Scaling method (UQ-bits)	M20K blocks	ALMs	Registers	DSPs
Nearest neighbor (UQ16.16)	12 / 2713	357 / 427 200	583	1 / 1518
Blinear (UQ16.16)	48 / 2713	733 / 427 200	1768	10 / 1518

Table 7.4: Resource usage on Intel Arria 10 GX 1150

As seen from Table 7.4 nearest-neighbor interpolation has a very low resource usage on the FPGA. Only one DSP was used, and this is because one DSP on the Intel Arria 10 FPGA can support two 18x18 multiplications [12]. Given that nearest-neighbor only has two 18x18 multiplications, used for calculating x- and y-values in the reverse mapping algorithm, these calculations share the same DSP. The number of registers used and the amount of M20K memory taken up by the design is also very low, and thus we can conclude with that this design is very resource efficient.

The same is seen for the bilinear results. Given the vastly more complex computations in regards to nearest-neighbor, the resource usage is still low. The register usage is only 3 times that of nearest-neighbor, and only 0.17% of the total available resources on the FPGA is used. The memory configuration uses 4 times the amount of nearest-neighbor, but this is still only 1.8% of the total M20K memory available. DSP usage is quite a lot higher for the bilinear calculation, but this is expected given the vastly more complicated calculations.

Given these results, we can safely conclude with that both the nearest-neighbor and the bilinear interpolation implementations were a success in regards to performance and resource usage. Because the implementations were not completed in regards to use the controller and scaler wrapper to control the scaler implementations, the resource usage is expected to go a bit up if these are properly implemented. It is not expected to go up by a great margin, and given the results from Table 7.1, the performance is also expected to stay roughly the same.

Chapter 8

Conclusion and Future Work

8.1 Video Scaler

In this project a video scaler was implemented in VHDL. The scaler supported upscaling up to a resolution of 3840x2160 using nearest-neighbor and bilinear interpolation. The synthesis test showed that the design was capable of running above the 300 MHz target frequency, which translates to an ability to support up to 4k 30fps video running at 248 MHz serial data transmission. To be able to support 4k 60fps video, two scalers of this design needs to be run in parallel.

The nearest-neighbor interpolation did work as intended with a very low resource usage taking up less than 1% of the available resources on an Intel Arria 10 GX 1150 FPGA. Unfortunately there were a unresolved bug in the bilinear interpolation algorithm prohibiting the design from working correctly using a 4-line framebuffer. The design therefore had to use a full-sized framebuffer equal to the size of an input frame. However, the synthesis tests showed that this design also would have had a very low resource usage at around 1% of an Intel Arria 10 GX 1150.

The objective image quality test showed that the performance of this scaler was lower, in terms of image quality, compared to MATLABs built-in scaler for both nearest-neighbor and bilinear interpolation. The reason for this was most likely the fixed-point representation used together with the truncation rounding method in the design. However, subjective visual comparisons between the two showed a very subtle, or no difference at all, giving the impression that the bad results seen from the VHDL implementation was a result of shifting of the pixel positions, not wrong pixel value calculations.

8.2 Avalon-ST VIP

UVVM was used in this project as a basis for the verification of the design. An Avalon-ST Verification IP was implemented, as there were no existing implementation of this in the UVVM community. The VIP was used for sub-module testing of the scaler design, as the top level of the scaler were not completed due to time limitations. The Avalon-ST VIP did function as intended, and it will be uploaded to the UVVM community with a MIT licence for public use and further development.

8.3 Future Work

Since the top level of the scaler were not completed, this is something that would be wanted to be further developed in a future implementation. This would make the scaler usable in an actual system, as the current implementation only works in simulations. Further development of the scaler to support dynamic resolutions for scaling would also be a desire. This way the scaler would not need to be re-compiled to support a different resolution. Optimizing the design to allow for two scalers to run in parallel would also add the possibility to support 4k 60fps video.

An interesting thing would also be to use 27x27 multiplication DSPs instead of the 18x18 DSPs used in this design. Using these 27x27 DSPs could improve the accuracy of the fixed-point representations, which might in turn improve on the objective image quality results. An implementation using normal rounding to the nearest integer in stead of truncation could also help with this, but this would require a re-design of the framebuffer memory address calculations for this to work.

Finally, support for bicubic interpolation and dynamic switching between the different interpolation algorithms would be desirable. Bicubic interpolation uses many of the same concepts as bilinear interpolation, so the bilinear implementation could be used as a basis for a future bicubic implementation.

Bibliography

[1] S. A. Fahmy, "Generalised parallel bilinear interpolation architecture for vision systems," in *2008 International Conference on Reconfigurable Computing and FPGAs*, Dec 2008, pp. 331–336.

[2] E. Tallaksen, *UVVM — The Fastest Growing FPGA Verification Methodology Worldwide!*, March 2019, Workshop on Open-Source Design Automation (OSDA). [Online]. Available: https://osda.gitlab.io/19/tallaksen.pdf

[3] BBC Natural History Unit, *Planet Earth II.* British Broadcasting Corporation (BBC), 2016, Screengrab from 4K UHD Blu-Ray release using FFmpeg.

[4] T. Stenseth, "Exploration of video scaling algorithms for FPGA implementation," December 2018, Project Thesis, NTNU.

[5] Intel, *Avalon Interface Specifications*, September 2018, Document version 2018.09.26. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[6] ——, *Video and Image Processing Suite User Guide*, September 2018, Document version 2018.09.24. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf

[7] D. G. Bailey, *Design for Embedded Image Processing on FPGAs.* Solaris South Tower, Singapore: John Wiley & Sons (Asia) Pte Ltd, 2011.

[8] C. C. Lin, M. H. Sheu, H. K. Chiang, C. Liaw, Z. C. Wu and W. K. Tsai, "An efficient architecture of extended linear interpolation for image processing," *Journal of Information Science and Engineering*, vol. 26, pp. 631–648, March 2010.

[9] R. G. Keys, "Cubic convolution interpolation for digital image processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 6, pp. 1153–1160, December 1981.

[10] A. Finnerty and H. Ratigner, *Reduce Power and Cost by Converting from Floating Point to Fixed Point*, March 2017, white paper published by Xilinx Inc, WP491 (v1.0). [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf

[11] Intel, *Intel Arria 10 Device Overview*, September 2018, Document version 2018.12.06. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf

[12] ——, *Intel Arria 10 Native Fixed Point DSP IP Core User Guide*, March 2017, Document version 2017.03.13. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nfp_dsp.pdf

[13] Bitvis AS, “UVVM v2018.12.03,” 2016–2018. [Online]. Available: https://github.com/UVVM/UVVM

[14] Walt Disney Feature Animation, *Lion King*. Walt Disney Pictures, 1994, Screengrab from Blu-Ray release using FFmpeg.

[15] Pixar Animation Studios, *Toy Story*. Walt Disney Pictures, 1995, Screengrab from Blu-Ray release using FFmpeg.

[16] FFmpeg Developers, “FFmpeg v4.0.2,” 2000–2018. [Online]. Available: https://www.ffmpeg.org/

Appendix A

VHDL source code

A.1 FIFO

```
1  --------------------------------------------------------
2  -- Project: FPGA video scaler
3  -- Author: Thomas Stenseth
4  -- Date: 2019-03-11
5  -- Version: 0.1
6  --------------------------------------------------------
7  -- Description: Generic single clock FIFO with
8  --              almostfull, full and empty signals
9  --------------------------------------------------------
10
11
12 library ieee;
13 use ieee.std_logic_1164.all;
14 use ieee.numeric_std.all;
15
16
17 entity fifo_generic is
18    generic (
19       g_width         : natural   := 8;
20       g_depth         : natural   := 32;
21       g_ramstyle      : string    := "MLAB";
22       g_output_reg    : boolean   := false
23    );
24    port (
25       clk_i           : in  std_logic;
26       sreset_i        : in  std_logic;
27       -- Write
28       data_i          : in  std_logic_vector(g_width-1 downto 0);
29       wr_en_i         : in  std_logic;
30       full_o          : out std_logic := '0';
31       almostfull_o    : out std_logic := '0';
32       -- Read
33       data_o          : out std_logic_vector(g_width-1 downto 0) := (others => '0');
34       rd_en_i         : in  std_logic;
35       empty_o         : out std_logic := '1'
36    );
37 end fifo_generic;
38
39 architecture fifo_generic_arc of fifo_generic is
40    -- RAM
41    type t_ram is array (natural range <>) of std_logic_vector(g_width-1 downto 0);
42    signal ram_data       : t_ram(g_depth-1 downto 0)                := (others => (others => '0'));
43    signal ram_out        : std_logic_vector(g_width-1 downto 0)     := (others => '0');
44    signal ram_out_reg    : std_logic_vector(g_width-1 downto 0)     := (others => '0');
45
46    -- RAM style
47    attribute ramstyle : string;
48    attribute ramstyle of ram_data : signal is g_ramstyle;
49
50    -- Signals
51    signal ram_wr_ptr     : integer range 0 to g_depth-1;   -- RAM write pointer
52    signal ram_rd_ptr     : integer range 0 to g_depth-1;   -- RAM read pointer
```

```
53      signal words_in_ram  : integer range 0 to g_depth;
54
55      signal wr_ok         : std_logic := '0';
56      signal rd_ok         : std_logic := '0';
57      signal is_full       : std_logic := '0';
58      signal is_almostfull : std_logic := '0';
59      signal is_empty      : std_logic := '1';
60   begin
61      -- Validate write and read
62      wr_ok <= '1' when wr_en_i = '1' and is_full = '0' else '0';
63      rd_ok <= '1' when rd_en_i = '1' and is_empty = '0' else '0';
64
65      -- Update number of words in ram
66      p_words : process(clk_i) is
67      begin
68         if rising_edge(clk_i) then
69            if sreset_i = '1' then
70               words_in_ram   <= 0;
71            else
72               -- FIFO write
73               if (wr_ok = '1' and rd_ok = '0') then
74                  words_in_ram <= words_in_ram + 1;
75               -- FIFO read
76               elsif (wr_ok = '0' and rd_ok = '1') then
77                  words_in_ram <= words_in_ram - 1;
78               -- FIFO both read and write, or no action
79               else
80                  words_in_ram <= words_in_ram;
81               end if;
82            end if;
83         end if;
84      end process p_words;
85
86      -- Update empty and full signals
87      p_flags : process(clk_i) is
88      begin
89         if rising_edge(clk_i) then
90            if sreset_i = '1' then
91               is_empty <= '1';
92               is_full <= '0';
93            else
94               -- Assert empty signal
95               if (words_in_ram = 0) or (words_in_ram = 1 and wr_ok = '0' and rd_ok = '1') then
96                  is_empty <= '1';
97               else
98                  is_empty <= '0';
99               end if;
100              -- Assert full signal
101              if(words_in_ram = g_depth) or (words_in_ram = g_depth-1 and wr_ok = '1' and rd_ok = '0') then
102                 is_full <= '1';
103              else
104                 is_full <= '0';
105              end if;
106              -- Assert almostfull signal
107              if(words_in_ram = g_depth) or (words_in_ram = g_depth-1) or (words_in_ram = g_depth-2 and wr_ok = '1' and
                   ↪    rd_ok = '0') then
108                 is_almostfull <= '1';
109              else
110                 is_almostfull <= '0';
111              end if;
112           end if;
113        end if;
114     end process p_flags;
115
116     -- Update write pointer
117     p_ram_wr_ptr : process(clk_i) is
118     begin
119        if rising_edge(clk_i) then
120           if sreset_i = '1' then
121              ram_wr_ptr <= 0;
122           elsif wr_ok = '1' then
123              ram_wr_ptr <= (ram_wr_ptr + 1) mod g_depth;
124           end if;
125        end if;
126     end process p_ram_wr_ptr;
127
128     -- Update read pointer
129     p_ram_rd_ptr : process(clk_i) is
130     begin
131        if rising_edge(clk_i) then
132           if sreset_i = '1' then
133              ram_rd_ptr <= 0;
134           elsif rd_ok = '1' then
135              ram_rd_ptr <= (ram_rd_ptr + 1) mod g_depth;
136           end if;
137        end if;
138     end process p_ram_rd_ptr;
139
140     -- Write to FIFO
141     p_write : process(clk_i) is
```

```
142       begin
143          if rising_edge(clk_i) then
144             if wr_ok = '1' then
145                ram_data(ram_wr_ptr) <= data_i;
146             end if;
147          end if;
148       end process p_write;
149
150       -- Read from FIFO
151       p_read : process(clk_i) is
152       begin
153          if rising_edge(clk_i) then
154             if rd_ok = '1' then
155                ram_out <= ram_data(ram_rd_ptr);
156             end if;
157             if g_output_reg then
158                ram_out_reg <= ram_out;
159             end if;
160          end if;
161       end process p_read;
162
163       -- Outputs
164       full_o          <= is_full;
165       almostfull_o    <= is_almostfull;
166       empty_o         <= is_empty;
167       data_o          <= ram_out_reg when g_output_reg else ram_out;
168
169    end fifo_generic_arc;
```

A.2 Simple Dual-Port RAM

```
1  --------------------------------------------------------
2  -- Project: FPGA video scaler
3  -- Author: Thomas Stenseth
4  -- Date: 2019-03-11
5  -- Version: 0.1
6  --------------------------------------------------------
7  -- Description: Simple dual-port RAM
8  --------------------------------------------------------
9
10
11 library ieee;
12 use ieee.std_logic_1164.all;
13 use ieee.numeric_std.all;
14
15
16 entity simple_dpram is
17    generic (
18       g_ram_width     : natural    := 8;
19       g_ram_depth     : natural    := 32;
20       g_ramstyle      : string     := "M20K";
21       g_output_reg    : boolean    := false
22    );
23    port (
24       clk_i        : in std_logic;
25       -- Write
26       data_i       : in std_logic_vector(g_ram_width-1 downto 0);
27       wr_addr_i    : in integer range 0 to g_ram_depth-1;
28       wr_en_i      : in std_logic;
29       -- Read
30       data_o       : out std_logic_vector(g_ram_width-1 downto 0) := (others => '0');
31       rd_addr_i    : in integer range 0 to g_ram_depth-1
32    );
33
34 end simple_dpram;
35
36 architecture rtl of simple_dpram is
37    -- RAM
38    type t_ram is array (natural range <>) of std_logic_vector(g_ram_width-1 downto 0);
39    signal ram_data       : t_ram(g_ram_depth-1 downto 0)                := (others => (others => '0'));
40    signal ram_out        : std_logic_vector(g_ram_width-1 downto 0)    := (others => '0');
41    signal ram_out_reg    : std_logic_vector(g_ram_width-1 downto 0)    := (others => '0');
42
43    -- RAM style
44    attribute ramstyle : string;
45    attribute ramstyle of ram_data : signal is g_ramstyle;
46 begin
47
48    p_ram : process(clk_i)
49    begin
50       if(rising_edge(clk_i)) then
51          -- Write to RAM
52          if(wr_en_i = '1') then
53             ram_data(wr_addr_i) <= data_i;
54          end if;
55
56          -- Read from RAM
57          ram_out      <= ram_data(rd_addr_i);
58          ram_out_reg <= ram_out;
59       end if;
60    end process p_ram;
61
62    -- Outputs
63    data_o <= ram_out_reg when g_output_reg else ram_out;
64
65 end rtl;
```

A.3 Multiport RAM

```vhdl
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-03-11
-- Version: 0.1
--------------------------------------------------------
-- Description: Multiport RAM consisting of
--                4 simple dual-port RAMs
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multiport_ram is
   generic (
      g_ram_width     : natural    := 8;
      g_ram_depth     : natural    := 32;
      g_ramstyle      : string     := "M20K";
      g_output_reg    : boolean    := false
   );
   port (
      clk_i           : in std_logic;
      -- Write
      data_i          : in std_logic_vector(g_ram_width-1 downto 0);
      wr_addr_i       : in integer range 0 to g_ram_depth-1;
      wr_en_i         : in std_logic;
      -- Read
      data_a_o        : out std_logic_vector(g_ram_width-1 downto 0) := (others => '0');
      data_b_o        : out std_logic_vector(g_ram_width-1 downto 0) := (others => '0');
      data_c_o        : out std_logic_vector(g_ram_width-1 downto 0) := (others => '0');
      data_d_o        : out std_logic_vector(g_ram_width-1 downto 0) := (others => '0');
      rd_addr_a_i     : in integer range 0 to g_ram_depth-1;
      rd_addr_b_i     : in integer range 0 to g_ram_depth-1;
      rd_addr_c_i     : in integer range 0 to g_ram_depth-1;
      rd_addr_d_i     : in integer range 0 to g_ram_depth-1
   );
end multiport_ram;

architecture rtl of multiport_ram is
   constant C_NUM_PORTS     : integer := 4;

   type t_read_addr is array(0 to C_NUM_PORTS-1) of integer range 0 to g_ram_depth-1;
   type t_read_data is array(0 to C_NUM_PORTS-1) of std_logic_vector(g_ram_width-1 downto 0);

   signal read_addr : t_read_addr;
   signal read_data : t_read_data;

   component simple_dpram
      generic (
         g_ram_width     : natural    := 8;
         g_ram_depth     : natural    := 32;
         g_ramstyle      : string     := "M20K";
         g_output_reg    : boolean    := false
      );
      port (
         clk_i        : in std_logic;
         -- Write
         data_i       : in std_logic_vector(g_ram_width-1 downto 0);
         wr_addr_i    : in integer range 0 to g_ram_depth-1;
         wr_en_i      : in std_logic;
         -- Read
         data_o       : out std_logic_vector(g_ram_width-1 downto 0) := (others => '0');
         rd_addr_i    : in integer range 0 to g_ram_depth-1
      );
   end component;

begin

   g_multiport_ram : for i in 0 to C_NUM_PORTS-1 generate
      u_simple_dpram : simple_dpram
      generic map (
         g_ram_width     => g_ram_width,
         g_ram_depth     => g_ram_depth,
         g_ramstyle      => "M20K",
         g_output_reg    => true
      )
      port map(
         clk_i           => clk_i,
         -- Write
         data_i          => data_i,
         wr_addr_i       => wr_addr_i,
         wr_en_i         => wr_en_i,
         -- Read
         data_o          => read_data(i),
```

```
87          rd_addr_i       => read_addr(i)
88        );
89    end generate g_multiport_ram;
90
91    data_a_o <= read_data(0);
92    data_b_o <= read_data(1);
93    data_c_o <= read_data(2);
94    data_d_o <= read_data(3);
95
96    read_addr(0) <= rd_addr_a_i;
97    read_addr(1) <= rd_addr_b_i;
98    read_addr(2) <= rd_addr_c_i;
99    read_addr(3) <= rd_addr_d_i;
100
101 end rtl;
```

A.4 My Fixed Package

```
-------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-04-24
-- Version: 0.1
-------------------------------------------------------
-- Description: Custom version of fixed point package
--              using fixed_truncate round style
--              instead of the default fixed_round
-------------------------------------------------------


library ieee;
use ieee.fixed_float_types.all;


package my_fixed_pkg is new ieee.fixed_generic_pkg
   generic map (
      fixed_round_style    => fixed_truncate,
      fixed_overflow_style => ieee.fixed_float_types.fixed_saturate,
      fixed_guard_bits     => 3,
      no_warning           => false
   );
end package my_fixed_pkg;
```

A.5 Nearest-Neighbor Scaling

```vhdl
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-04-14
-- Version: 0.1
--------------------------------------------------------
-- Description: Nearest-neighbor interpolation
--                using 4-line framebuffer
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.my_fixed_pkg.all;

entity scaler is
   generic (
      g_data_width          : natural;
      g_rx_video_width      : natural;
      g_rx_video_height     : natural;
      g_tx_video_width      : natural;
      g_tx_video_height     : natural
   );
   port (
      clk_i                    : in  std_logic;
      sreset_i                 : in  std_logic;

      scaler_startofpacket_i   : in  std_logic;
      scaler_endofpacket_i     : in  std_logic;
      scaler_data_i            : in  std_logic_vector(g_data_width-1 downto 0);
      scaler_valid_i           : in  std_logic;
      scaler_ready_o           : out std_logic := '0';

      scaler_startofpacket_o   : out std_logic := '0';
      scaler_endofpacket_o     : out std_logic := '0';
      scaler_data_o            : out std_logic_vector(g_data_width-1 downto 0) := (others => '0');
      scaler_valid_o           : out std_logic := '0';
      scaler_ready_i           : in  std_logic
   );
end scaler;

architecture scaler_arc of scaler is
   type t_state is (s_idle, s_pre_fill_fb, s_finish_fill_fb, s_upscale, s_upscale_and_fill);
   signal state : t_state := s_idle;

   constant C_LINE_BUFFERS : integer := 4;

   -- Scaling ratio
   -- Using (others => '1') or else division by 0 error
   signal scaling_ratio        : ufixed(3 downto -12) := (others => '0');
   signal scaling_ratio_reg    : ufixed(3 downto -12) := (others => '0');

   signal tx_height            : ufixed(11 downto 0) := (others => '1');
   signal rx_height            : ufixed(11 downto 0) := (others => '1');
   signal tx_height_reg        : ufixed(11 downto 0) := (others => '1');
   signal rx_height_reg        : ufixed(11 downto 0) := (others => '1');

   -- Framebuffer
   signal fb_wr_en_i        : std_logic := '0';
   signal fb_wr_en_reg      : std_logic := '0';
   signal fb_data_i         : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_data_reg       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_wr_addr_i      : integer := 0;
   signal fb_wr_addr_reg    : integer := 0;
   signal fb_valid_reg      : std_logic := '0';
   signal fb_data_o         : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_rd_addr_i      : integer := 0;

   -- Scaler
   signal interpolate       : boolean := false;

   -- Mapping function
   signal dx            : ufixed(16 downto -16) := (others => '0');
   signal dy            : ufixed(16 downto -16) := (others => '0');
   signal dx_reg        : ufixed(16 downto -16) := (others => '0');
   signal dy_reg        : ufixed(16 downto -16) := (others => '0');

   signal dx_1          : ufixed(15 downto -12) := (others => '0');
   signal dy_1          : ufixed(15 downto -12) := (others => '0');
   signal dx_1_reg      : ufixed(15 downto -12) := (others => '0');
   signal dy_1_reg      : ufixed(15 downto -12) := (others => '0');

   signal dxy_2           : ufixed(15 downto -16) := (others => '0');
   signal dxy_2_reg       : ufixed(15 downto -16) := (others => '0');
```

```
87
88      signal dx_int         : integer := 0;
89      signal dy_int         : integer := 0;
90      signal dx_int_reg     : integer := 0;
91      signal dy_int_reg     : integer := 0;
92
93      signal x_count           : integer := 0;
94      signal y_count           : integer := 0;
95      signal x_count_ufx       : ufixed(11 downto 0) := (others => '0');
96      signal y_count_ufx       : ufixed(11 downto 0) := (others => '0');
97      signal x_count_ufx_reg   : ufixed(11 downto 0) := (others => '0');
98      signal y_count_ufx_reg   : ufixed(11 downto 0) := (others => '0');
99
100     signal dy_int_last    : integer := 0;
101     signal dy_change      : boolean := false;
102
103     -- Counters
104     signal cur_input      : integer := 0;
105     signal cur_output     : integer := 0;
106
107
108 begin
109     framebuffer : entity work.simple_dpram
110     generic map (
111        g_ram_width    => g_data_width,
112        g_ram_depth    => g_rx_video_width*C_LINE_BUFFERS,
113        g_ramstyle     => "M20K",
114        g_output_reg   => true
115     )
116     port map(
117        clk_i          => clk_i,
118        -- Write
119        data_i         => fb_data_i,
120        wr_addr_i      => fb_wr_addr_i,
121        wr_en_i        => fb_wr_en_i,
122        -- Read
123        data_o         => fb_data_o,
124        rd_addr_i      => fb_rd_addr_i
125     );
126
127
128
129     p_fsm : process(clk_i) is
130        variable v_count : integer := 0;
131     begin
132        if rising_edge(clk_i) then
133           case(state) is
134
135              when s_idle =>
136                 scaler_ready_o       <= '1';
137                 cur_input            <= 0;
138                 cur_output           <= 0;
139                 scaler_endofpacket_o <= '0';
140                 fb_valid_reg         <= '0';
141
142                 if scaler_ready_o = '1' and scaler_valid_i = '1' then
143                    if scaler_startofpacket_i = '1' then
144                       fb_wr_en_reg   <= '1';
145                       state          <= s_pre_fill_fb;
146                    end if;
147                 end if;
148
149
150              when s_pre_fill_fb =>
151                 -- Pre-fill framebuffer before starting the scaler
152                 if scaler_ready_o = '1' and scaler_valid_i = '1' then
153                    if fb_wr_addr_reg = (g_rx_video_width*C_LINE_BUFFERS)-2 then
154                       -- Ready latency of 1 on Avalon ST-video
155                       scaler_ready_o <= '0';
156                       fb_wr_en_reg   <= '1';
157                       fb_wr_addr_reg <= fb_wr_addr_reg + 1;
158                       cur_input      <= cur_input + 1;
159                       state          <= s_finish_fill_fb;
160                    else
161                       -- Fill framebuffer
162                       scaler_ready_o <= '1';
163                       fb_wr_en_reg   <= '1';
164                       fb_wr_addr_reg <= fb_wr_addr_reg + 1;
165                       cur_input      <= cur_input + 1;
166                    end if;
167                 end if;
168
169
170              when s_finish_fill_fb =>
171                 -- Fill the last data recieved after ready latency of 1
172                 if scaler_valid_i = '1' then
173                    fb_wr_en_reg   <= '0';
174                    fb_wr_addr_reg <= 0 when (fb_wr_addr_reg = (g_rx_video_width*C_LINE_BUFFERS)-1) else fb_wr_addr_reg +
                     ↪    1;
175                    cur_input      <= cur_input + 1;
```

```
176
177                 -- Upscaling
178                 state <= s_upscale;
179                 if interpolate = true then
180                    cur_output     <= cur_output + 1;
181                    fb_valid_reg   <= '1';
182                 end if;
183              end if;
184
185
186           when s_upscale =>
187              -- Upscaling process
188              if scaler_ready_i = '1' then
189                 interpolate    <= true;
190                 cur_output     <= cur_output + 1;
191                 scaler_ready_o <= '0';
192                 fb_wr_en_reg   <= '0';
193
194                 if cur_output >= 11 then
195                    -- First data on output
196                    -- Need +11 because delay through scaler is 11 clock cycles
197                    fb_valid_reg <= '1';
198                    scaler_startofpacket_o <= '1' when cur_output = 12 else '0';
199                 end if;
200
201
202                 if dy_change and (cur_input < (g_rx_video_width*g_rx_video_height)) then
203                    -- One line in framebuffer has been processed, ready to be refilled
204                    scaler_ready_o <= '1';
205                    fb_wr_en_reg   <= '1';
206                    interpolate    <= false;
207                    state          <= s_upscale_and_fill;
208                 end if;
209
210                 if cur_output >= (g_tx_video_width*g_tx_video_height)+6 then
211                    -- Done processing
212                    interpolate <= false;
213                 end if;
214
215                 if cur_output >= (g_tx_video_width*g_tx_video_height)+9 then
216                    -- Last data on output
217                    fb_valid_reg   <= '0';
218                    scaler_endofpacket_o <= '1';
219                    state          <= s_idle;
220                 end if;
221              else
222                 interpolate <= false;
223              end if;
224
225
226           when s_upscale_and_fill =>
227              -- Fill one line in framebuffer while upscaling
228              if scaler_ready_o = '1' and scaler_valid_i = '1' then
229                 if scaler_ready_i = '1' then
230                    interpolate    <= true;
231                    scaler_ready_o <= '1';
232                    fb_wr_en_reg   <= '1';
233                    fb_wr_addr_reg <= 0 when (fb_wr_addr_reg = (g_rx_video_width*C_LINE_BUFFERS)-1) else
                     ↪    fb_wr_addr_reg + 1;
234                    cur_input      <= cur_input + 1;
235                    fb_valid_reg <= '0';
236
237                    v_count := v_count + 1;
238                    if v_count >= 3 then
239                       -- 2 clock cycles delay from fb_rd_addr is set to data is on output
240                       fb_valid_reg   <= '1';
241                       cur_output     <= cur_output + 1;
242                    end if;
243
244                    if v_count = g_rx_video_width-1 then
245                       -- One line has been filled.
246                       -- Ready latency of 1 on Avalon ST-video
247                       scaler_ready_o <= '0';
248                       v_count        := 0;
249                       state          <= s_finish_fill_fb;
250                    end if;
251                 else
252                    interpolate <= false;
253                 end if;
254              end if;
255
256        end case;
257
258        -- Connect registers
259        fb_wr_en_i     <= fb_wr_en_reg;
260        fb_wr_addr_i   <= fb_wr_addr_reg;
261        fb_data_i      <= scaler_data_i;
262        scaler_valid_o <= fb_valid_reg;
263
264        -- Handle reset
```

```
265            if sreset_i = '1' then
266               state <= s_idle;
267            end if;
268         end if;
269      end process p_fsm;
270
271
272
273      p_nearest : process(clk_i) is
274      begin
275         if rising_edge(clk_i) then
276            if interpolate then
277               -- Make x/y_count ufixed
278               x_count_ufx       <= to_ufixed(x_count, x_count_ufx);
279               y_count_ufx       <= to_ufixed(y_count, y_count_ufx);
280               x_count_ufx_reg   <= x_count_ufx;
281               y_count_ufx_reg   <= y_count_ufx;
282
283               -- Fixed point DSP multiplication of variable part of dx/dy calculation
284               dx_1     <= x_count_ufx_reg * scaling_ratio_reg;
285               dy_1     <= y_count_ufx_reg * scaling_ratio_reg;
286               dx_1_reg <= dx_1;
287               dy_1_reg <= dy_1;
288
289               -- Constant part of dx/dy calculation
290               dxy_2       <= to_ufixed(0.5, 1, -2) * (1 - resize(scaling_ratio_reg, 12, -14));
291               dxy_2_reg   <= dxy_2;
292
293               -- Final dx/dy calculation
294               dx       <= dx_1_reg + dxy_2_reg;
295               dy       <= dy_1_reg + dxy_2_reg;
296               dx_reg   <= dx;
297               dy_reg   <= dy;
298
299               -- Next pixel in target frame
300               x_count <= x_count + 1;
301
302               -- Check if a row in target frame is completed
303               if x_count = g_tx_video_width-1 then
304                  x_count <= 0;
305                  y_count <= y_count + 1;
306               end if;
307
308               -- Check if all rowns in line buffer is completed
309               if dy_reg >= C_LINE_BUFFERS then
310                  -- Reset y_count for frambuffer addresses
311                  y_count           <= 0;
312                  y_count_ufx       <= to_ufixed(0, y_count_ufx);
313                  y_count_ufx_reg   <= to_ufixed(0, y_count_ufx_reg);
314                  -- Variable part of dx/dy is zero, use only constant part
315                  dy                <= resize(dxy_2_reg, dy'high, dy'low);
316                  dy_reg            <= resize(dxy_2_reg, dy'high, dy'low);
317                  dy_int            <= 0;
318                  -- Unable to set dx_1/dy_1 to zero, as this ruins fixed point DSP implementation
319                  -- This is instead handled by setting y_count_ufx/y_count_ufx_reg to zero.
320                  -- This will give wrong dx_1/dy_1 calculation on current clock cycle,
321                  -- but this is fixed by forcing dy_int <= 0 and having y_count set to zero.
322               else
323                  dy_int <= to_integer(dy_reg);
324               end if;
325
326               -- Use floor from my_fixed_pkg to get dx/dy to integer for fb_rd_addr
327               dx_int      <= to_integer(dx_reg);
328               dx_int_reg  <= dx_int;
329               dy_int_reg  <= dy_int;
330
331               -- Find nearest neighbor address for framebuffer
332               fb_rd_addr_i <= g_rx_video_width*dy_int_reg + dx_int_reg;
333
334               -- Check if scaler is done with a framebuffer line
335               dy_int_last <= dy_int;
336               dy_change   <= true when dy_int_last /= dy_int else false;
337            end if;
338
339            scaler_data_o <= fb_data_o;
340         end if;
341      end process p_nearest;
342
343
344
345      p_scaling_ratio : process(clk_i) is
346      begin
347         if rising_edge(clk_i) then
348            -- Calc scaling ratio
349            -- Needs to be inside clocked process to become registers for fixed point DSP implementation
350            rx_height         <= to_ufixed(g_rx_video_height, rx_height);
351            tx_height         <= to_ufixed(g_tx_video_height, tx_height);
352            rx_height_reg     <= rx_height;
353            tx_height_reg     <= tx_height;
354            scaling_ratio     <= resize(rx_height_reg/tx_height_reg, scaling_ratio'high, scaling_ratio'low);
```

```
355            scaling_ratio_reg <= scaling_ratio;
356        end if;
357     end process p_scaling_ratio;
358
359  end scaler_arc;
```

A.6 Bilinear Scaling 4-line Framebuffer

```
1   -------------------------------------------------------
2   -- Project: FPGA video scaler
3   -- Author: Thomas Stenseth
4   -- Date: 2019-04-14
5   -- Version: 0.1
6   -------------------------------------------------------
7   -- Description: Bilinear interpolation using
8   --               4-line framebuffer
9   -------------------------------------------------------
10
11
12  library ieee;
13  use ieee.std_logic_1164.all;
14  use ieee.numeric_std.all;
15
16  use work.my_fixed_pkg.all;
17
18  entity scaler is
19     generic (
20        g_data_width         : natural;
21        g_rx_video_width     : natural;
22        g_rx_video_height    : natural;
23        g_tx_video_width     : natural;
24        g_tx_video_height    : natural
25     );
26     port (
27        clk_i              : in  std_logic;
28        sreset_i           : in  std_logic;
29
30        scaler_startofpacket_i  : in  std_logic;
31        scaler_endofpacket_i    : in  std_logic;
32        scaler_data_i           : in  std_logic_vector(g_data_width-1 downto 0);
33        scaler_valid_i          : in  std_logic;
34        scaler_ready_o          : out std_logic := '0';
35
36        scaler_startofpacket_o  : out std_logic := '0';
37        scaler_endofpacket_o    : out std_logic := '0';
38        scaler_data_o           : out std_logic_vector(g_data_width-1 downto 0) := (others => '0');
39        scaler_valid_o          : out std_logic := '0';
40        scaler_ready_i          : in  std_logic
41     );
42  end scaler;
43
44  architecture scaler_arc of scaler is
45     type t_state is (s_idle, s_pre_fill_fb, s_finish_fill_fb, s_upscale, s_upscale_and_fill);
46     signal state : t_state := s_idle;
47
48     constant C_LINE_BUFFERS : integer := 4;
49
50     -- Scaling ratio
51     -- Using (others => '1') or else division by 0 error
52     signal scaling_ratio       : ufixed(3 downto -12) := (others => '0');
53     signal scaling_ratio_reg   : ufixed(3 downto -12) := (others => '0');
54
55     signal tx_height           : ufixed(11 downto 0) := (others => '1');
56     signal rx_height           : ufixed(11 downto 0) := (others => '1');
57     signal tx_height_reg       : ufixed(11 downto 0) := (others => '1');
58     signal rx_height_reg       : ufixed(11 downto 0) := (others => '1');
59
60     -- Framebuffer
61     signal fb_wr_en_i        : std_logic := '0';
62     signal fb_wr_en_reg      : std_logic := '0';
63     signal fb_data_i         : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
64     signal fb_data_reg       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
65     signal fb_wr_addr_i      : integer := 0;
66     signal fb_wr_addr_reg    : integer := 0;
67     signal fb_valid_reg      : std_logic := '0';
68     signal fb_data_a_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
69     signal fb_data_b_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
70     signal fb_data_c_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
71     signal fb_data_d_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
72     signal fb_rd_addr_a_i    : integer := 0;
73     signal fb_rd_addr_b_i    : integer := 0;
74     signal fb_rd_addr_c_i    : integer := 0;
75     signal fb_rd_addr_d_i    : integer := 0;
76
77     -- Scaler
78     signal interpolate       : boolean := false;
79
80     -- Mapping function
81     signal dx            : ufixed(16 downto -16) := (others => '0');
82     signal dy            : ufixed(16 downto -16) := (others => '0');
83     signal dx_reg        : ufixed(16 downto -16) := (others => '0');
84     signal dy_reg        : ufixed(16 downto -16) := (others => '0');
85
86     signal dx_1          : ufixed(15 downto -12) := (others => '0');
```

```
87      signal dy_1           : ufixed(15 downto -12)  := (others => '0');
88      signal dx_1_reg       : ufixed(15 downto -12)  := (others => '0');
89      signal dy_1_reg       : ufixed(15 downto -12)  := (others => '0');
90
91      signal dxy_2          : ufixed(15 downto -16)  := (others => '0');
92      signal dxy_2_reg      : ufixed(15 downto -16)  := (others => '0');
93
94      -- Needs to be 1 because of dx/dy algorithm
95      signal x_count           : integer := 1;
96      signal y_count           : integer := 1;
97      signal x_count_ufx       : ufixed(11 downto 0)  := 12x"1";
98      signal y_count_ufx       : ufixed(11 downto 0)  := 12x"1";
99      signal x_count_ufx_reg   : ufixed(11 downto 0)  := 12x"1";
100     signal y_count_ufx_reg   : ufixed(11 downto 0)  := 12x"1";
101
102
103     signal x1_int         : integer := 1;
104     signal x2_int         : integer := 2;
105     signal y1_int         : integer := 1;
106     signal y2_int         : integer := 2;
107     signal pix1_int       : integer := 0;
108     signal pix2_int       : integer := 0;
109     signal pix3_int       : integer := 0;
110     signal pix4_int       : integer := 0;
111
112     signal dy_int         : integer := 1;
113     signal dy_int_last    : integer := 1;
114     signal dy_change      : boolean := false;
115
116     -- Delays
117     signal dx_reg_1       : ufixed(16 downto -16)  := (others => '0');
118     signal dy_reg_1       : ufixed(16 downto -16)  := (others => '0');
119     signal dx_reg_2       : ufixed(16 downto -16)  := (others => '0');
120     signal dy_reg_2       : ufixed(16 downto -16)  := (others => '0');
121     signal dx_reg_3       : ufixed(16 downto -16)  := (others => '0');
122     signal dy_reg_3       : ufixed(16 downto -16)  := (others => '0');
123     signal dx_reg_4       : ufixed(16 downto -16)  := (others => '0');
124     signal dy_reg_4       : ufixed(16 downto -16)  := (others => '0');
125     signal dx_reg_5       : ufixed(16 downto -16)  := (others => '0');
126     signal dy_reg_5       : ufixed(16 downto -16)  := (others => '0');
127     signal dx_reg_6       : ufixed(16 downto -16)  := (others => '0');
128     signal dy_reg_6       : ufixed(16 downto -16)  := (others => '0');
129     signal dx_reg_7       : ufixed(16 downto -16)  := (others => '0');
130     signal dy_reg_7       : ufixed(16 downto -16)  := (others => '0');
131
132     signal x1_int_reg_1   : integer := 0;
133     signal x2_int_reg_1   : integer := 0;
134     signal y1_int_reg_1   : integer := 0;
135     signal y2_int_reg_1   : integer := 0;
136     signal x1_int_reg_2   : integer := 0;
137     signal x2_int_reg_2   : integer := 0;
138     signal y1_int_reg_2   : integer := 0;
139     signal y2_int_reg_2   : integer := 0;
140     signal x1_int_reg_3   : integer := 0;
141     signal x2_int_reg_3   : integer := 0;
142     signal y1_int_reg_3   : integer := 0;
143     signal y2_int_reg_3   : integer := 0;
144     signal x1_int_reg_4   : integer := 0;
145     signal x2_int_reg_4   : integer := 0;
146     signal y1_int_reg_4   : integer := 0;
147     signal y2_int_reg_4   : integer := 0;
148     signal x1_int_reg_5   : integer := 0;
149     signal x2_int_reg_5   : integer := 0;
150     signal y1_int_reg_5   : integer := 0;
151     signal y2_int_reg_5   : integer := 0;
152     signal x1_int_reg_6   : integer := 0;
153     signal x2_int_reg_6   : integer := 0;
154     signal y1_int_reg_6   : integer := 0;
155     signal y2_int_reg_6   : integer := 0;
156
157     -- Coefficients
158     signal delta_x1       : ufixed(1 downto -16)  := (others => '0');
159     signal delta_x2       : ufixed(1 downto -16)  := (others => '0');
160     signal delta_y1       : ufixed(1 downto -16)  := (others => '0');
161     signal delta_y2       : ufixed(1 downto -16)  := (others => '0');
162     signal delta_x1_reg   : ufixed(1 downto -16)  := (others => '0');
163     signal delta_x2_reg   : ufixed(1 downto -16)  := (others => '0');
164     signal delta_y1_reg   : ufixed(1 downto -16)  := (others => '0');
165     signal delta_y2_reg   : ufixed(1 downto -16)  := (others => '0');
166
167     signal delta_y1_reg_1  : ufixed(1 downto -16)  := (others => '0');
168     signal delta_y2_reg_1  : ufixed(1 downto -16)  := (others => '0');
169     signal delta_y1_reg_2  : ufixed(1 downto -16)  := (others => '0');
170     signal delta_y2_reg_2  : ufixed(1 downto -16)  := (others => '0');
171     signal delta_y1_reg_3  : ufixed(1 downto -16)  := (others => '0');
172     signal delta_y2_reg_3  : ufixed(1 downto -16)  := (others => '0');
173     signal delta_y1_reg_4  : ufixed(1 downto -16)  := (others => '0');
174     signal delta_y2_reg_4  : ufixed(1 downto -16)  := (others => '0');
175
176     signal pix1_data_ufx : ufixed(g_data_width-1 downto 0)   := (others => '0');
```

```
177     signal pix2_data_ufx : ufixed(g_data_width-1 downto 0)    := (others => '0');
178     signal pix3_data_ufx : ufixed(g_data_width-1 downto 0)    := (others => '0');
179     signal pix4_data_ufx : ufixed(g_data_width-1 downto 0)    := (others => '0');
180
181     signal pix1_data_ufx_reg : ufixed(g_data_width-1 downto 0)    := (others => '0');
182     signal pix2_data_ufx_reg : ufixed(g_data_width-1 downto 0)    := (others => '0');
183     signal pix3_data_ufx_reg : ufixed(g_data_width-1 downto 0)    := (others => '0');
184     signal pix4_data_ufx_reg : ufixed(g_data_width-1 downto 0)    := (others => '0');
185
186     signal A_y1_a         : ufixed(9 downto -16) := (others => '0');
187     signal A_y1_b         : ufixed(9 downto -16) := (others => '0');
188     signal A_y2_a         : ufixed(9 downto -16) := (others => '0');
189     signal A_y2_b         : ufixed(9 downto -16) := (others => '0');
190     signal A_y1           : ufixed(7 downto -8) := (others => '0');
191     signal A_y2           : ufixed(7 downto -8) := (others => '0');
192     signal A_1            : ufixed(7 downto -8) := (others => '0');
193     signal A_2            : ufixed(7 downto -8) := (others => '0');
194     signal A              : ufixed(7 downto 0) := (others => '0');
195
196     signal B_y1_a         : ufixed(9 downto -16) := (others => '0');
197     signal B_y1_b         : ufixed(9 downto -16) := (others => '0');
198     signal B_y2_a         : ufixed(9 downto -16) := (others => '0');
199     signal B_y2_b         : ufixed(9 downto -16) := (others => '0');
200     signal B_y1           : ufixed(7 downto -8) := (others => '0');
201     signal B_y2           : ufixed(7 downto -8) := (others => '0');
202     signal B_1            : ufixed(7 downto -8) := (others => '0');
203     signal B_2            : ufixed(7 downto -8) := (others => '0');
204     signal B              : ufixed(7 downto 0) := (others => '0');
205
206     signal C_y1_a         : ufixed(9 downto -16) := (others => '0');
207     signal C_y1_b         : ufixed(9 downto -16) := (others => '0');
208     signal C_y2_a         : ufixed(9 downto -16) := (others => '0');
209     signal C_y2_b         : ufixed(9 downto -16) := (others => '0');
210     signal C_y1           : ufixed(7 downto -8) := (others => '0');
211     signal C_y2           : ufixed(7 downto -8) := (others => '0');
212     signal C_1            : ufixed(7 downto -8) := (others => '0');
213     signal C_2            : ufixed(7 downto -8) := (others => '0');
214     signal C              : ufixed(7 downto 0) := (others => '0');
215
216     signal A_y1_a_reg     : ufixed(9 downto -16) := (others => '0');
217     signal A_y1_b_reg     : ufixed(9 downto -16) := (others => '0');
218     signal A_y2_a_reg     : ufixed(9 downto -16) := (others => '0');
219     signal A_y2_b_reg     : ufixed(9 downto -16) := (others => '0');
220     signal A_y1_reg       : ufixed(7 downto -8) := (others => '0');
221     signal A_y2_reg       : ufixed(7 downto -8) := (others => '0');
222     signal A_1_reg        : ufixed(7 downto -8) := (others => '0');
223     signal A_2_reg        : ufixed(7 downto -8) := (others => '0');
224     signal A_reg          : ufixed(7 downto 0) := (others => '0');
225
226     signal B_y1_a_reg     : ufixed(9 downto -16) := (others => '0');
227     signal B_y1_b_reg     : ufixed(9 downto -16) := (others => '0');
228     signal B_y2_a_reg     : ufixed(9 downto -16) := (others => '0');
229     signal B_y2_b_reg     : ufixed(9 downto -16) := (others => '0');
230     signal B_y1_reg       : ufixed(7 downto -8) := (others => '0');
231     signal B_y2_reg       : ufixed(7 downto -8) := (others => '0');
232     signal B_1_reg        : ufixed(7 downto -8) := (others => '0');
233     signal B_2_reg        : ufixed(7 downto -8) := (others => '0');
234     signal B_reg          : ufixed(7 downto 0) := (others => '0');
235
236     signal C_y1_a_reg     : ufixed(9 downto -16) := (others => '0');
237     signal C_y1_b_reg     : ufixed(9 downto -16) := (others => '0');
238     signal C_y2_a_reg     : ufixed(9 downto -16) := (others => '0');
239     signal C_y2_b_reg     : ufixed(9 downto -16) := (others => '0');
240     signal C_y1_reg       : ufixed(7 downto -8) := (others => '0');
241     signal C_y2_reg       : ufixed(7 downto -8) := (others => '0');
242     signal C_1_reg        : ufixed(7 downto -8) := (others => '0');
243     signal C_2_reg        : ufixed(7 downto -8) := (others => '0');
244     signal C_reg          : ufixed(7 downto 0) := (others => '0');
245
246     -- Counters
247     signal cur_input      : integer := 0;
248     signal cur_output     : integer := 0;
249
250
251  begin
252     framebuffer : entity work.multiport_ram
253     generic map (
254        g_ram_width     => g_data_width,
255        g_ram_depth     => g_rx_video_width*C_LINE_BUFFERS,
256        g_ramstyle      => "M20K",
257        g_output_reg    => true
258     )
259     port map(
260        clk_i           => clk_i,
261        -- Write
262        data_i          => fb_data_i,
263        wr_addr_i       => fb_wr_addr_i,
264        wr_en_i         => fb_wr_en_i,
265        -- Read
266        data_a_o        => fb_data_a_o,
```

```
267          data_b_o        => fb_data_b_o,
268          data_c_o        => fb_data_c_o,
269          data_d_o        => fb_data_d_o,
270          rd_addr_a_i     => fb_rd_addr_a_i,
271          rd_addr_b_i     => fb_rd_addr_b_i,
272          rd_addr_c_i     => fb_rd_addr_c_i,
273          rd_addr_d_i     => fb_rd_addr_d_i
274       );
275
276
277
278       p_fsm : process(clk_i) is
279          variable v_count : integer := 0;
280       begin
281          if rising_edge(clk_i) then
282             case(state) is
283
284                when s_idle =>
285                   scaler_ready_o       <= '1';
286                   cur_input            <= 0;
287                   cur_output           <= 0;
288                   scaler_endofpacket_o <= '0';
289                   fb_valid_reg         <= '0';
290
291                   if scaler_ready_o = '1' and scaler_valid_i = '1' then
292                      if scaler_startofpacket_i = '1' then
293                         fb_wr_en_reg   <= '1';
294                         state          <= s_pre_fill_fb;
295                      end if;
296                   end if;
297
298
299                when s_pre_fill_fb =>
300                   -- Pre-fill framebuffer before starting the scaler
301                   if scaler_ready_o = '1' and scaler_valid_i = '1' then
302                      if fb_wr_addr_reg = (g_rx_video_width*C_LINE_BUFFERS)-2 then
303                         -- Ready latency of 1 on Avalon ST-video
304                         scaler_ready_o <= '0';
305                         fb_wr_en_reg   <= '1';
306                         fb_wr_addr_reg <= fb_wr_addr_reg + 1;
307                         cur_input      <= cur_input + 1;
308                         state          <= s_finish_fill_fb;
309                      else
310                         -- Fill framebuffer
311                         scaler_ready_o <= '1';
312                         fb_wr_en_reg   <= '1';
313                         fb_wr_addr_reg <= fb_wr_addr_reg + 1;
314                         cur_input      <= cur_input + 1;
315                      end if;
316                   end if;
317
318
319                when s_finish_fill_fb =>
320                   -- Fill the last data recieved after ready latency of 1
321                   if scaler_valid_i = '1' then
322                      fb_wr_en_reg   <= '0';
323                      fb_wr_addr_reg <= 0 when (fb_wr_addr_reg = (g_rx_video_width*C_LINE_BUFFERS)-1) else fb_wr_addr_reg +
                       ↪    1;
324                      cur_input      <= cur_input + 1;
325
326                      -- Upscaling
327                      state <= s_upscale;
328                      if interpolate = true then
329                         cur_output     <= cur_output + 1;
330                         fb_valid_reg   <= '1';
331                      end if;
332                   end if;
333
334
335                when s_upscale =>
336                   -- Upscaling process
337                   if scaler_ready_i = '1' then
338                      interpolate    <= true;
339                      cur_output     <= cur_output + 1;
340                      scaler_ready_o <= '0';
341                      fb_wr_en_reg   <= '0';
342
343                      if cur_output >= 18 then
344                         -- First data on output
345                         -- Need +18 because delay through scaler is 18 clock cycles
346                         fb_valid_reg <= '1';
347                         scaler_startofpacket_o <= '1' when cur_output = 19 else '0';
348                      end if;
349
350
351                      if dy_change and (cur_input < (g_rx_video_width*g_rx_video_height)) then
352                         -- One line in framebuffer has been processed, ready to be refilled
353                         scaler_ready_o <= '1';
354                         fb_wr_en_reg   <= '1';
355                         interpolate    <= false;
```

```
356                    state          <= s_upscale_and_fill;
357                end if;
358
359                if cur_output >= (g_tx_video_width*g_tx_video_height)+24 then
360                    -- Done processing
361                    -- +N depends on latency through scaler
362                    interpolate <= false;
363                end if;
364
365                if cur_output >= (g_tx_video_width*g_tx_video_height)+25 then
366                    -- Last data on output
367                    -- +N+1 depends on latency through scaler
368                    fb_valid_reg    <= '0';
369                    scaler_endofpacket_o <= '1';
370                    state          <= s_idle;
371                end if;
372            else
373                interpolate <= false;
374            end if;
375
376
377        when s_upscale_and_fill =>
378            -- Fill one line in framebuffer while upscaling
379            if scaler_ready_o = '1' and scaler_valid_i = '1' then
380                if scaler_ready_i = '1' then
381                    interpolate     <= true;
382                    scaler_ready_o <= '1';
383                    fb_wr_en_reg    <= '1';
384                    fb_wr_addr_reg <= 0 when (fb_wr_addr_reg = (g_rx_video_width*C_LINE_BUFFERS)-1) else
                     ↪    fb_wr_addr_reg + 1;
385                    cur_input      <= cur_input + 1;
386                    fb_valid_reg <= '0';
387
388                    v_count := v_count + 1;
389                    if v_count >= 3 then
390                        -- 2 clock cycles delay from fb_rd_addr is set to data is on output
391                        fb_valid_reg    <= '1';
392                        cur_output      <= cur_output + 1;
393                    end if;
394
395                    if v_count = g_rx_video_width-1 then
396                        -- One line has been filled.
397                        -- Ready latency of 1 on Avalon ST-video
398                        scaler_ready_o <= '0';
399                        v_count         := 0;
400                        state           <= s_finish_fill_fb;
401                    end if;
402                else
403                    interpolate <= false;
404                end if;
405            end if;
406
407    end case;
408
409    -- Connect registers
410    fb_wr_en_i      <= fb_wr_en_reg;
411    fb_wr_addr_i    <= fb_wr_addr_reg;
412    fb_data_i       <= scaler_data_i;
413    scaler_valid_o <= fb_valid_reg;
414
415    -- Handle reset
416    if sreset_i = '1' then
417        state <= s_idle;
418    end if;
419    end if;
420 end process p_fsm;
421
422
423
424 p_reverse_mapping : process(clk_i) is
425 begin
426    if rising_edge(clk_i) then
427        if interpolate then
428            -- Make x/y_count ufixed
429            x_count_ufx       <= to_ufixed(x_count, x_count_ufx);
430            y_count_ufx       <= to_ufixed(y_count, x_count_ufx);
431            x_count_ufx_reg   <= x_count_ufx;
432            y_count_ufx_reg   <= y_count_ufx;
433
434            -- Fixed point DSP multiplication of variable part of dx/dy calculation
435            dx_1      <= x_count_ufx_reg * scaling_ratio_reg;
436            dy_1      <= y_count_ufx_reg * scaling_ratio_reg;
437            dx_1_reg <= dx_1;
438            dy_1_reg <= dy_1;
439
440            -- Constant part of dx/dy calculation
441            dxy_2       <= to_ufixed(0.5, 1, -2) * (1 - resize(scaling_ratio_reg, 12, -14));
442            dxy_2_reg   <= dxy_2;
443
444            -- Final dx/dy calculation
```

```
445                 dx       <= dx_1_reg + dxy_2_reg;
446                 dy       <= dy_1_reg + dxy_2_reg;
447                 dx_reg   <= dx;
448                 dy_reg   <= dy;
449
450                 -- Next pixel in target frame
451                 x_count <= x_count + 1;
452
453                 -- Check if a row in target frame is completed
454                 if x_count = g_tx_video_width then
455                    x_count <= 1;
456                    y_count <= y_count + 1;
457                 end if;
458
459                 -- Keep kernel within boundaries
460                 if dx_reg < 1 then
461                    x1_int <= 1;
462                    x2_int <= 2;
463                    dx_reg_1 <= to_ufixed(1, dx_reg);
464                 elsif dx_reg > g_rx_video_width then
465                    x1_int <= g_rx_video_width - 1;
466                    x2_int <= g_rx_video_width;
467                    dx_reg_1 <= to_ufixed(g_rx_video_width, dx_reg);
468                 else
469                    x1_int <= to_integer(dx_reg);
470                    x2_int <= to_integer(dx_reg) + 1;
471                    dx_reg_1 <= dx_reg;
472                 end if;
473
474                 -- Keep kernel within boundaries
475                 if dy_reg < 1 then
476                    dy_int   <= 1;
477                    y1_int   <= 1;
478                    y2_int   <= 2;
479                    dy_reg_1 <= to_ufixed(1, dy_reg);
480                 elsif dy_reg >= C_LINE_BUFFERS+1 then
481                    -- Start from beginning of framebuffer when both lines have been completed
482                    y_count           <= 1;
483                    y_count_ufx       <= to_ufixed(1, y_count_ufx);
484                    y_count_ufx_reg   <= to_ufixed(1, y_count_ufx_reg);
485                    dy                <= resize(scaling_ratio_reg + dxy_2_reg, dy);
486                    dy_reg            <= resize(scaling_ratio_reg + dxy_2_reg, dy);
487                    dy_int            <= 1;
488                    y1_int            <= 1;
489                    y2_int            <= 2;
490                    dy_reg_1          <= to_ufixed(1, dy_reg);
491                 elsif dy_reg >= C_LINE_BUFFERS then
492                    -- Special case when one line has completed but not the other one
493                    dy_int   <= C_LINE_BUFFERS;
494                    y1_int   <= C_LINE_BUFFERS;
495                    y2_int   <= 1;
496                    dy_reg_1 <= dy_reg;
497                 else
498                    dy_int   <= to_integer(dy_reg);
499                    y1_int   <= to_integer(dy_reg);
500                    y2_int   <= to_integer(dy_reg) + 1;
501                    dy_reg_1 <= dy_reg;
502                 end if;
503
504
505                 -- Calculate framebuffer addresses for each pixel
506                 pix1_int <= ((y1_int-1)*g_rx_video_width) + (x1_int - 1);
507                 pix2_int <= ((y1_int-1)*g_rx_video_width) + (x2_int - 1);
508                 pix3_int <= ((y2_int-1)*g_rx_video_width) + (x1_int - 1);
509                 pix4_int <= ((y2_int-1)*g_rx_video_width) + (x2_int - 1);
510
511                 -- Read data from framebuffer
512                 fb_rd_addr_a_i <= pix1_int;
513                 fb_rd_addr_b_i <= pix2_int;
514                 fb_rd_addr_c_i <= pix3_int;
515                 fb_rd_addr_d_i <= pix4_int;
516
517                 -- Get pixel values
518                 pix1_data_ufx <= to_ufixed(fb_data_a_o, pix1_data_ufx);
519                 pix2_data_ufx <= to_ufixed(fb_data_b_o, pix2_data_ufx);
520                 pix3_data_ufx <= to_ufixed(fb_data_c_o, pix3_data_ufx);
521                 pix4_data_ufx <= to_ufixed(fb_data_d_o, pix4_data_ufx);
522
523                 pix1_data_ufx_reg <= pix1_data_ufx;
524                 pix2_data_ufx_reg <= pix2_data_ufx;
525                 pix3_data_ufx_reg <= pix3_data_ufx;
526                 pix4_data_ufx_reg <= pix4_data_ufx;
527
528                 -- Delays
529                 -- TODO: Implement as shift register
530                 dx_reg_2 <= dx_reg_1;
531                 dy_reg_2 <= dy_reg_1;
532                 dx_reg_3 <= dx_reg_2;
533                 dy_reg_3 <= dy_reg_2;
534                 dx_reg_4 <= dx_reg_3;
```

```
535             dy_reg_4 <= dy_reg_3;
536             dx_reg_5 <= dx_reg_4;
537             dy_reg_5 <= dy_reg_4;
538             dx_reg_6 <= dx_reg_5;
539             dy_reg_6 <= dy_reg_5;
540             dx_reg_7 <= dx_reg_6;
541             dy_reg_7 <= dy_reg_6;
542
543             x1_int_reg_1 <= x1_int;
544             x2_int_reg_1 <= x2_int;
545             y1_int_reg_1 <= y1_int;
546             y2_int_reg_1 <= y2_int;
547             x1_int_reg_2 <= x1_int_reg_1;
548             x2_int_reg_2 <= x2_int_reg_1;
549             y1_int_reg_2 <= y1_int_reg_1;
550             y2_int_reg_2 <= y2_int_reg_1;
551             x1_int_reg_3 <= x1_int_reg_2;
552             x2_int_reg_3 <= x2_int_reg_2;
553             y1_int_reg_3 <= y1_int_reg_2;
554             y2_int_reg_3 <= y2_int_reg_2;
555             x1_int_reg_4 <= x1_int_reg_3;
556             x2_int_reg_4 <= x2_int_reg_3;
557             y1_int_reg_4 <= y1_int_reg_3;
558             y2_int_reg_4 <= y2_int_reg_3;
559             x1_int_reg_5 <= x1_int_reg_4;
560             x2_int_reg_5 <= x2_int_reg_4;
561             y1_int_reg_5 <= y1_int_reg_4;
562             y2_int_reg_5 <= y2_int_reg_4;
563             x1_int_reg_6 <= x1_int_reg_5;
564             x2_int_reg_6 <= x2_int_reg_5;
565             y1_int_reg_6 <= y1_int_reg_5;
566             y2_int_reg_6 <= y2_int_reg_5;
567
568
569             -- Calculate deltas
570             delta_x1 <= resize(dx_reg_6 - x1_int_reg_5, delta_x1);
571             delta_x2 <= resize(x2_int_reg_5 - dx_reg_6, delta_x2);
572
573             if y1_int_reg_5 = C_LINE_BUFFERS and y2_int_reg_5 = 1 then
574                -- Special case
575                delta_y1 <= resize(dy_reg_6 - y1_int_reg_5, delta_y1);
576                delta_y2 <= resize((y2_int_reg_5 + C_LINE_BUFFERS) - dy_reg_6, delta_y2);
577             else
578                delta_y1 <= resize(dy_reg_6 - y1_int_reg_5, delta_y1);
579                delta_y2 <= resize(y2_int_reg_5 - dy_reg_6, delta_y2);
580             end if;
581
582             -- Registers
583             delta_x1_reg   <= delta_x1;
584             delta_x2_reg   <= delta_x2;
585             delta_y1_reg   <= delta_y1;
586             delta_y2_reg   <= delta_y2;
587
588             -- Delay
589             -- TODO: Implement as shift register
590             delta_y1_reg_1 <= delta_y1_reg;
591             delta_y2_reg_1 <= delta_y2_reg;
592             delta_y1_reg_2 <= delta_y1_reg_1;
593             delta_y2_reg_2 <= delta_y2_reg_1;
594             delta_y1_reg_3 <= delta_y1_reg_2;
595             delta_y2_reg_3 <= delta_y2_reg_2;
596             delta_y1_reg_4 <= delta_y1_reg_3;
597             delta_y2_reg_4 <= delta_y2_reg_3;
598
599             -- Calculate pixel values
600             A_y1_a <= delta_x2_reg*pix1_data_ufx_reg(7 downto 0);
601             A_y1_b <= delta_x1_reg*pix2_data_ufx_reg(7 downto 0);
602             A_y2_a <= delta_x2_reg*pix3_data_ufx_reg(7 downto 0);
603             A_y2_b <= delta_x1_reg*pix4_data_ufx_reg(7 downto 0);
604
605             B_y1_a <= delta_x2_reg*pix1_data_ufx_reg(15 downto 8);
606             B_y1_b <= delta_x1_reg*pix2_data_ufx_reg(15 downto 8);
607             B_y2_a <= delta_x2_reg*pix3_data_ufx_reg(15 downto 8);
608             B_y2_b <= delta_x1_reg*pix4_data_ufx_reg(15 downto 8);
609
610             C_y1_a <= delta_x2_reg*pix1_data_ufx_reg(23 downto 16);
611             C_y1_b <= delta_x1_reg*pix2_data_ufx_reg(23 downto 16);
612             C_y2_a <= delta_x2_reg*pix3_data_ufx_reg(23 downto 16);
613             C_y2_b <= delta_x1_reg*pix4_data_ufx_reg(23 downto 16);
614
615             -- Registers
616             A_y1_a_reg <= A_y1_a;
617             A_y1_b_reg <= A_y1_b;
618             A_y2_a_reg <= A_y2_a;
619             A_y2_b_reg <= A_y2_b;
620             B_y1_a_reg <= B_y1_a;
621             B_y1_b_reg <= B_y1_b;
622             B_y2_a_reg <= B_y2_a;
623             B_y2_b_reg <= B_y2_b;
624             C_y1_a_reg <= C_y1_a;
```

```
625             C_y1_b_reg <= C_y1_b;
626             C_y2_a_reg <= C_y2_a;
627             C_y2_b_reg <= C_y2_b;
628
629             A_y1 <= resize(A_y1_a_reg + A_y1_b_reg, A_y1);
630             A_y2 <= resize(A_y2_a_reg + A_y2_b_reg, A_y2);
631
632             B_y1 <= resize(B_y1_a_reg + B_y1_b_reg, B_y1);
633             B_y2 <= resize(B_y2_a_reg + B_y2_b_reg, B_y2);
634
635             C_y1 <= resize(C_y1_a_reg + C_y1_b_reg, C_y1);
636             C_y2 <= resize(C_y2_a_reg + C_y2_b_reg, C_y2);
637
638             -- Registers
639             A_y1_reg <= A_y1;
640             A_y2_reg <= A_y2;
641             B_y1_reg <= B_y1;
642             B_y2_reg <= B_y2;
643             C_y1_reg <= C_y1;
644             C_y2_reg <= C_y2;
645
646             A_1 <= resize(delta_y2_reg_4*A_y1_reg, A_1);
647             A_2 <= resize(delta_y1_reg_4*A_y2_reg, A_2);
648
649             B_1 <= resize(delta_y2_reg_4*B_y1_reg, B_1);
650             B_2 <= resize(delta_y1_reg_4*B_y2_reg, B_2);
651
652             C_1 <= resize(delta_y2_reg_4*C_y1_reg, C_1);
653             C_2 <= resize(delta_y1_reg_4*C_y2_reg, C_2);
654
655             -- Registers
656             A_1_reg <= A_1;
657             A_2_reg <= A_2;
658             B_1_reg <= B_1;
659             B_2_reg <= B_2;
660             C_1_reg <= C_1;
661             C_2_reg <= C_2;
662
663             -- Final calculation of new pixel value
664             A  <= resize(A_1_reg + A_2_reg, A);
665             B  <= resize(B_1_reg + B_2_reg, B);
666             C  <= resize(C_1_reg + C_2_reg, C);
667
668             A_reg <= A;
669             B_reg <= B;
670             C_reg <= C;
671
672             -- Check if scaler is done with a framebuffer line
673             dy_int_last <= dy_int;
674             dy_change   <= true when dy_int_last /= dy_int else false;
675          end if;
676
677          scaler_data_o(7 downto 0)     <= std_logic_vector(unsigned(A_reg));
678          scaler_data_o(15 downto 8)    <= std_logic_vector(unsigned(B_reg));
679          scaler_data_o(23 downto 16)   <= std_logic_vector(unsigned(C_reg));
680
681          -- Handle reset
682          if sreset_i = '1' then
683             x_count <= 1; -- Needs to be 1 because of dx/dy algorithm
684             y_count <= 1; -- Needs to be 1 because of dx/dy algorithm
685          end if;
686       end if;
687    end process p_reverse_mapping;
688
689    p_scaling_ratio : process(clk_i) is
690    begin
691       if rising_edge(clk_i) then
692          -- Calc scaling ratio
693          -- Needs to be inside clocked process to become registers for fixed point DSP implementation
694          rx_height         <= to_ufixed(g_rx_video_height, rx_height);
695          tx_height         <= to_ufixed(g_tx_video_height, tx_height);
696          rx_height_reg     <= rx_height;
697          tx_height_reg     <= tx_height;
698          scaling_ratio     <= resize(rx_height_reg/tx_height_reg, scaling_ratio'high, scaling_ratio'low);
699          scaling_ratio_reg <= scaling_ratio;
700       end if;
701    end process p_scaling_ratio;
702
703 end scaler_arc;
```

A.7 Bilinear Scaling Full Size Framebuffer

```vhdl
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-04-14
-- Version: 0.1
--------------------------------------------------------
-- Description: Bilinear interpolation using
--              framebuffer the size of rx video
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.my_fixed_pkg.all;

entity scaler is
   generic (
      g_data_width         : natural;
      g_rx_video_width     : natural;
      g_rx_video_height    : natural;
      g_tx_video_width     : natural;
      g_tx_video_height    : natural
   );
   port (
      clk_i              : in  std_logic;
      sreset_i           : in  std_logic;

      scaler_startofpacket_i  : in  std_logic;
      scaler_endofpacket_i    : in  std_logic;
      scaler_data_i           : in  std_logic_vector(g_data_width-1 downto 0);
      scaler_valid_i          : in  std_logic;
      scaler_ready_o          : out std_logic := '0';

      scaler_startofpacket_o  : out std_logic := '0';
      scaler_endofpacket_o    : out std_logic := '0';
      scaler_data_o           : out std_logic_vector(g_data_width-1 downto 0) := (others => '0');
      scaler_valid_o          : out std_logic := '0';
      scaler_ready_i          : in  std_logic
   );
end scaler;

architecture scaler_arc of scaler is
   type t_state is (s_idle, s_pre_fill_fb, s_finish_fill_fb, s_upscale);
   signal state : t_state := s_idle;

   -- Scaling ratio
   -- Using (others => '1') or else division by 0 error
   signal scaling_ratio       : ufixed(3 downto -12) := (others => '0');
   signal scaling_ratio_reg   : ufixed(3 downto -12) := (others => '0');

   signal tx_height           : ufixed(11 downto 0) := (others => '1');
   signal rx_height           : ufixed(11 downto 0) := (others => '1');
   signal tx_height_reg       : ufixed(11 downto 0) := (others => '1');
   signal rx_height_reg       : ufixed(11 downto 0) := (others => '1');

   -- Framebuffer
   signal fb_wr_en_i        : std_logic := '0';
   signal fb_wr_en_reg      : std_logic := '0';
   signal fb_data_i         : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_data_reg       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_wr_addr_i      : integer := 0;
   signal fb_wr_addr_reg    : integer := 0;
   signal fb_valid_reg      : std_logic := '0';
   signal fb_data_a_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_data_b_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_data_c_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_data_d_o       : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal fb_rd_addr_a_i    : integer := 0;
   signal fb_rd_addr_b_i    : integer := 0;
   signal fb_rd_addr_c_i    : integer := 0;
   signal fb_rd_addr_d_i    : integer := 0;

   -- Scaler
   signal interpolate       : boolean := false;

   -- Mapping function
   signal dx             : ufixed(16 downto -16) := (others => '0');
   signal dy             : ufixed(16 downto -16) := (others => '0');
   signal dy_fb          : ufixed(16 downto -16) := (others => '0');
   signal dx_reg         : ufixed(16 downto -16) := (others => '0');
   signal dy_reg         : ufixed(16 downto -16) := (others => '0');
   signal dy_fb_reg      : ufixed(16 downto -16) := (others => '0');

   signal dx_reg_1       : ufixed(16 downto -16) := (others => '0');
```

```
87      signal dy_reg_1       : ufixed(16 downto -16) := (others => '0');
88
89      signal dx_1           : ufixed(15 downto -12) := (others => '0');
90      signal dy_1           : ufixed(15 downto -12) := (others => '0');
91      signal dy_fb_1        : ufixed(15 downto -12) := (others => '0');
92      signal dx_1_reg       : ufixed(15 downto -12) := (others => '0');
93      signal dy_1_reg       : ufixed(15 downto -12) := (others => '0');
94      signal dy_fb_1_reg    : ufixed(15 downto -12) := (others => '0');
95
96      signal dxy_2          : ufixed(15 downto -16) := (others => '0');
97      signal dxy_2_reg      : ufixed(15 downto -16) := (others => '0');
98
99      -- Needs to be 1 because of dx/dy algorithm
100     signal x_count           : integer := 1;
101     signal y_count           : integer := 1;
102     signal y_count_fb        : integer := 1;
103     signal x_count_ufx       : ufixed(11 downto 0) := 12x"1";
104     signal y_count_ufx       : ufixed(11 downto 0) := 12x"1";
105     signal y_count_fb_ufx    : ufixed(11 downto 0) := 12x"1";
106     signal x_count_ufx_reg   : ufixed(11 downto 0) := 12x"1";
107     signal y_count_ufx_reg   : ufixed(11 downto 0) := 12x"1";
108     signal y_count_fb_ufx_reg   : ufixed(11 downto 0) := 12x"1";
109
110
111     signal x1_int         : integer := 1;
112     signal x2_int         : integer := 2;
113     signal y1_int         : integer := 1;
114     signal y2_int         : integer := 2;
115     signal y1_fb_int      : integer := 1;
116     signal y2_fb_int      : integer := 2;
117
118     signal pix1_int       : integer := 0;
119     signal pix2_int       : integer := 0;
120     signal pix3_int       : integer := 0;
121     signal pix4_int       : integer := 0;
122     signal pix1_data      : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
123     signal pix2_data      : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
124     signal pix3_data      : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
125     signal pix4_data      : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
126
127     signal dy_int         : integer := 1;
128     signal dy_int_last    : integer := 1;
129     signal dy_change         : boolean := false;
130
131     -- Coefficients
132     signal delta1       : ufixed(1 downto -16) := (others => '0');
133     signal delta2       : ufixed(1 downto -16) := (others => '0');
134     signal delta3       : ufixed(1 downto -16) := (others => '0');
135     signal delta4       : ufixed(1 downto -16) := (others => '0');
136
137     signal delta1_reg_1   : ufixed(1 downto -16) := (others => '0');
138     signal delta2_reg_1   : ufixed(1 downto -16) := (others => '0');
139     signal delta3_reg_1   : ufixed(1 downto -16) := (others => '0');
140     signal delta4_reg_1   : ufixed(1 downto -16) := (others => '0');
141
142     signal delta1_reg_2   : ufixed(1 downto -16) := (others => '0');
143     signal delta2_reg_2   : ufixed(1 downto -16) := (others => '0');
144     signal delta3_reg_2   : ufixed(1 downto -16) := (others => '0');
145     signal delta4_reg_2   : ufixed(1 downto -16) := (others => '0');
146
147     signal delta1_reg_3   : ufixed(1 downto -16) := (others => '0');
148     signal delta2_reg_3   : ufixed(1 downto -16) := (others => '0');
149     signal delta3_reg_3   : ufixed(1 downto -16) := (others => '0');
150     signal delta4_reg_3   : ufixed(1 downto -16) := (others => '0');
151
152     signal delta1_reg_4   : ufixed(1 downto -16) := (others => '0');
153     signal delta2_reg_4   : ufixed(1 downto -16) := (others => '0');
154     signal delta3_reg_4   : ufixed(1 downto -16) := (others => '0');
155     signal delta4_reg_4   : ufixed(1 downto -16) := (others => '0');
156
157     signal pix1_data_A : ufixed(7 downto 0)    := (others => '0');
158     signal pix2_data_A : ufixed(7 downto 0)    := (others => '0');
159     signal pix3_data_A : ufixed(7 downto 0)    := (others => '0');
160     signal pix4_data_A : ufixed(7 downto 0)    := (others => '0');
161     signal pix1_data_B : ufixed(7 downto 0)    := (others => '0');
162     signal pix2_data_B : ufixed(7 downto 0)    := (others => '0');
163     signal pix3_data_B : ufixed(7 downto 0)    := (others => '0');
164     signal pix4_data_B : ufixed(7 downto 0)    := (others => '0');
165     signal pix1_data_C : ufixed(7 downto 0)    := (others => '0');
166     signal pix2_data_C : ufixed(7 downto 0)    := (others => '0');
167     signal pix3_data_C : ufixed(7 downto 0)    := (others => '0');
168     signal pix4_data_C : ufixed(7 downto 0)    := (others => '0');
169
170     signal A_y1_a         : ufixed(9 downto -16) := (others => '0');
171     signal A_y1_b         : ufixed(9 downto -16) := (others => '0');
172     signal A_y2_a         : ufixed(9 downto -16) := (others => '0');
173     signal A_y2_b         : ufixed(9 downto -16) := (others => '0');
174     signal A_y1           : ufixed(7 downto -8) := (others => '0');
175     signal A_y2           : ufixed(7 downto -8) := (others => '0');
176     signal A_1            : ufixed(7 downto -8) := (others => '0');
```

```
177      signal A_2           : ufixed(7 downto -8) := (others => '0');
178      signal A             : ufixed(7 downto 0) := (others => '0');
179
180      signal B_y1_a        : ufixed(9 downto -16) := (others => '0');
181      signal B_y1_b        : ufixed(9 downto -16) := (others => '0');
182      signal B_y2_a        : ufixed(9 downto -16) := (others => '0');
183      signal B_y2_b        : ufixed(9 downto -16) := (others => '0');
184      signal B_y1          : ufixed(7 downto -8) := (others => '0');
185      signal B_y2          : ufixed(7 downto -8) := (others => '0');
186      signal B_1           : ufixed(7 downto -8) := (others => '0');
187      signal B_2           : ufixed(7 downto -8) := (others => '0');
188      signal B             : ufixed(7 downto 0) := (others => '0');
189
190      signal C_y1_a        : ufixed(9 downto -16) := (others => '0');
191      signal C_y1_b        : ufixed(9 downto -16) := (others => '0');
192      signal C_y2_a        : ufixed(9 downto -16) := (others => '0');
193      signal C_y2_b        : ufixed(9 downto -16) := (others => '0');
194      signal C_y1          : ufixed(7 downto -8) := (others => '0');
195      signal C_y2          : ufixed(7 downto -8) := (others => '0');
196      signal C_1           : ufixed(7 downto -8) := (others => '0');
197      signal C_2           : ufixed(7 downto -8) := (others => '0');
198      signal C             : ufixed(7 downto 0) := (others => '0');
199
200      -- Counters
201      signal cur_input     : integer := 0;
202      signal cur_output    : integer := 0;
203
204
205   begin
206      framebuffer : entity work.multiport_ram
207      generic map (
208         g_ram_width     => g_data_width,
209         g_ram_depth     => g_rx_video_width*g_rx_video_height,
210         g_ramstyle      => "M20K",
211         g_output_reg    => true
212      )
213      port map(
214         clk_i           => clk_i,
215         -- Write
216         data_i          => fb_data_i,
217         wr_addr_i       => fb_wr_addr_i,
218         wr_en_i         => fb_wr_en_i,
219         -- Read
220         data_a_o        => fb_data_a_o,
221         data_b_o        => fb_data_b_o,
222         data_c_o        => fb_data_c_o,
223         data_d_o        => fb_data_d_o,
224         rd_addr_a_i     => fb_rd_addr_a_i,
225         rd_addr_b_i     => fb_rd_addr_b_i,
226         rd_addr_c_i     => fb_rd_addr_c_i,
227         rd_addr_d_i     => fb_rd_addr_d_i
228      );
229
230
231
232      p_fsm : process(clk_i) is
233         variable v_count : integer := 0;
234      begin
235         if rising_edge(clk_i) then
236            case(state) is
237
238               when s_idle =>
239                  scaler_ready_o       <= '1';
240                  cur_input            <= 0;
241                  cur_output           <= 0;
242                  scaler_endofpacket_o <= '0';
243                  fb_valid_reg         <= '0';
244
245                  if scaler_ready_o = '1' and scaler_valid_i = '1' then
246                     if scaler_startofpacket_i = '1' then
247                        fb_wr_en_reg    <= '1';
248                        state           <= s_pre_fill_fb;
249                     end if;
250                  end if;
251
252
253               when s_pre_fill_fb =>
254                  -- Pre-fill framebuffer before starting the scaler
255                  if scaler_ready_o = '1' and scaler_valid_i = '1' then
256                     if fb_wr_addr_reg = (g_rx_video_width*g_rx_video_height)-2 then
257                        -- Ready latency of 1 on Avalon ST-video
258                        scaler_ready_o <= '0';
259                        fb_wr_en_reg   <= '1';
260                        fb_wr_addr_reg <= fb_wr_addr_reg + 1;
261                        cur_input      <= cur_input + 1;
262                        state          <= s_finish_fill_fb;
263                     else
264                        -- Fill framebuffer
265                        scaler_ready_o <= '1';
266                        fb_wr_en_reg   <= '1';
```

```
267                     fb_wr_addr_reg <= fb_wr_addr_reg + 1;
268                     cur_input      <= cur_input + 1;
269                  end if;
270               end if;
271
272
273            when s_finish_fill_fb =>
274               -- Fill the last data recieved after ready latency of 1
275               if scaler_valid_i = '1' then
276                  fb_wr_en_reg   <= '0';
277                  fb_wr_addr_reg <= 0 when (fb_wr_addr_reg = (g_rx_video_width*g_rx_video_height)-1) else
                   ↪    fb_wr_addr_reg + 1;
278                  cur_input      <= cur_input + 1;
279
280                  -- Upscaling
281                  state <= s_upscale;
282               end if;
283
284
285            when s_upscale =>
286               -- Upscaling process
287               if scaler_ready_i = '1' then
288                  interpolate    <= true;
289                  cur_output     <= cur_output + 1;
290                  scaler_ready_o <= '0';
291                  fb_wr_en_reg   <= '0';
292
293                  if cur_output >= 13 then
294                     -- First data on output
295                     -- Need +13 because delay through scaler is 13 clock cycles
296                     fb_valid_reg <= '1';
297                     scaler_startofpacket_o <= '1' when cur_output = 14 else '0';
298                  end if;
299
300                  if cur_output >= (g_tx_video_width*g_tx_video_height)+24 then
301                     -- Done processing
302                     -- +N depends on latency through scaler
303                     interpolate <= false;
304                  end if;
305
306                  if cur_output >= (g_tx_video_width*g_tx_video_height)+25 then
307                     -- Last data on output
308                     -- +N+1 depends on latency through scaler
309                     fb_valid_reg   <= '0';
310                     scaler_endofpacket_o <= '1';
311                     state          <= s_idle;
312                  end if;
313               else
314                  interpolate <= false;
315               end if;
316
317         end case;
318
319         -- Connect registers
320         fb_wr_en_i     <= fb_wr_en_reg;
321         fb_wr_addr_i   <= fb_wr_addr_reg;
322         fb_data_i      <= scaler_data_i;
323         scaler_valid_o <= fb_valid_reg;
324
325         -- Handle reset
326         if sreset_i = '1' then
327            state <= s_idle;
328         end if;
329      end if;
330   end process p_fsm;
331
332
333
334   p_reverse_mapping : process(clk_i) is
335   begin
336      if rising_edge(clk_i) then
337         if interpolate then
338            -- Make x/y_count ufixed
339            x_count_ufx          <= to_ufixed(x_count, x_count_ufx);
340            y_count_ufx          <= to_ufixed(y_count, x_count_ufx);
341            y_count_fb_ufx       <= to_ufixed(y_count, x_count_ufx);
342            x_count_ufx_reg      <= x_count_ufx;
343            y_count_ufx_reg      <= y_count_ufx;
344            y_count_fb_ufx_reg   <= y_count_fb_ufx;
345
346            -- Fixed point DSP multiplication of variable part of dx/dy calculation
347            dx_1        <= x_count_ufx_reg * scaling_ratio_reg;
348            dy_1        <= y_count_ufx_reg * scaling_ratio_reg;
349            dy_fb_1     <= y_count_fb_ufx_reg * scaling_ratio_reg;
350            dx_1_reg    <= dx_1;
351            dy_1_reg    <= dy_1;
352            dy_fb_1_reg <= dy_fb_1;
353
354            -- Constant part of dx/dy calculation
355            dxy_2       <= to_ufixed(0.5, 1, -2) * (1 - resize(scaling_ratio_reg, 12, -14));
```

```
356             dxy_2_reg   <= dxy_2;
357
358             -- Final dx/dy calculation
359             dx          <= dx_1_reg + dxy_2_reg;
360             dy          <= dy_1_reg + dxy_2_reg;
361             dy_fb       <= dy_fb_1_reg + dxy_2_reg;
362             dx_reg      <= dx;
363             dy_reg      <= dy;
364             dy_fb_reg   <= dy_fb;
365
366             -- Next pixel in target frame
367             x_count <= x_count + 1;
368
369            -- Check if a row in target frame is completed
370             if x_count = g_tx_video_width then
371                x_count <= 1;
372                y_count <= y_count + 1;
373             end if;
374
375             -- Keep kernel within boundaries
376             if dx_reg < 1 then
377                x1_int   <= 1;
378                x2_int   <= 2;
379                dx_reg_1 <= to_ufixed(1, dx_reg);
380             elsif dx_reg > g_rx_video_width then
381                x1_int   <= g_rx_video_width - 1;
382                x2_int   <= g_rx_video_width;
383                dx_reg_1 <= to_ufixed(g_rx_video_width, dx_reg);
384             else
385                x1_int   <= to_integer(dx_reg);
386                x2_int   <= to_integer(dx_reg) + 1;
387                dx_reg_1 <= dx_reg;
388             end if;
389
390             -- Keep kernel within boundaries
391             if dy_reg < 1 then
392                y1_int   <= 1;
393                y2_int   <= 2;
394                dy_reg_1 <= to_ufixed(1, dy_reg);
395             elsif dy_reg > g_rx_video_height then
396                y1_int   <= g_rx_video_height - 1;
397                y2_int   <= g_rx_video_height;
398                dy_reg_1 <= to_ufixed(g_rx_video_width, dy_reg);
399             else
400                y1_int   <= to_integer(dy_reg);
401                y2_int   <= to_integer(dy_reg) + 1;
402                dy_reg_1 <= dy_reg;
403             end if;
404
405             -- Read data from framebuffer
406             fb_rd_addr_a_i <= ((y1_int-1)*g_rx_video_width) + (x1_int - 1);
407             fb_rd_addr_b_i <= ((y1_int-1)*g_rx_video_width) + (x2_int - 1);
408             fb_rd_addr_c_i <= ((y2_int-1)*g_rx_video_width) + (x1_int - 1);
409             fb_rd_addr_d_i <= ((y2_int-1)*g_rx_video_width) + (x2_int - 1);
410
411             pix1_data_A <= to_ufixed(fb_data_a_o(7 downto 0), pix1_data_A);
412             pix2_data_A <= to_ufixed(fb_data_b_o(7 downto 0), pix2_data_A);
413             pix3_data_A <= to_ufixed(fb_data_c_o(7 downto 0), pix3_data_A);
414             pix4_data_A <= to_ufixed(fb_data_d_o(7 downto 0), pix4_data_A);
415
416             pix1_data_B <= to_ufixed(fb_data_a_o(15 downto 8), pix1_data_B);
417             pix2_data_B <= to_ufixed(fb_data_b_o(15 downto 8), pix2_data_B);
418             pix3_data_B <= to_ufixed(fb_data_c_o(15 downto 8), pix3_data_B);
419             pix4_data_B <= to_ufixed(fb_data_d_o(15 downto 8), pix4_data_B);
420
421             pix1_data_C <= to_ufixed(fb_data_a_o(23 downto 16), pix1_data_C);
422             pix2_data_C <= to_ufixed(fb_data_b_o(23 downto 16), pix2_data_C);
423             pix3_data_C <= to_ufixed(fb_data_c_o(23 downto 16), pix3_data_C);
424             pix4_data_C <= to_ufixed(fb_data_d_o(23 downto 16), pix4_data_C);
425
426
427             delta1 <= resize(x2_int - dx_reg_1, delta1);
428             delta2 <= resize(dx_reg_1 - x1_int, delta2);
429             delta3 <= resize(y2_int - dy_reg_1, delta3);
430             delta4 <= resize(dy_reg_1 - y1_int, delta4);
431
432             -- Delay
433             -- TODO: Implement as shift register
434             delta1_reg_1 <= delta1;
435             delta2_reg_1 <= delta2;
436             delta3_reg_1 <= delta3;
437             delta4_reg_1 <= delta4;
438             delta1_reg_2 <= delta1_reg_1;
439             delta2_reg_2 <= delta2_reg_1;
440             delta3_reg_2 <= delta3_reg_1;
441             delta4_reg_2 <= delta4_reg_1;
442             delta1_reg_3 <= delta1_reg_2;
443             delta2_reg_3 <= delta2_reg_2;
444             delta3_reg_3 <= delta3_reg_2;
445             delta4_reg_3 <= delta4_reg_2;
```

```
446             delta1_reg_4 <= delta1_reg_3;
447             delta2_reg_4 <= delta2_reg_3;
448             delta3_reg_4 <= delta3_reg_3;
449             delta4_reg_4 <= delta4_reg_3;
450
451
452             -- Calculate pixel values
453             -- Warning:
454             -- Not pipelined, will not fit in a DSP
455             -- This as not been prioritized as the framebuffer will not fit on an FPGA anyway
456             A_y1 <= resize(delta1_reg_3*pix1_data_A + delta2_reg_3*pix2_data_A, A_y1);
457             A_y2 <= resize(delta1_reg_3*pix3_data_A + delta2_reg_3*pix4_data_A, A_y2);
458             A  <= resize(delta3_reg_4*A_y1 + delta4_reg_4*A_y2, A);
459
460             B_y1 <= resize(delta1_reg_3*pix1_data_B + delta2_reg_3*pix2_data_B, B_y1);
461             B_y2 <= resize(delta1_reg_3*pix3_data_B + delta2_reg_3*pix4_data_B, B_y2);
462             B  <= resize(delta3_reg_4*B_y1 + delta4_reg_4*B_y2, B);
463
464             C_y1 <= resize(delta1_reg_3*pix1_data_C + delta2_reg_3*pix2_data_C, C_y1);
465             C_y2 <= resize(delta1_reg_3*pix3_data_C + delta2_reg_3*pix4_data_C, C_y2);
466             C  <= resize(delta3_reg_4*C_y1 + delta4_reg_4*C_y2, C);
467
468          end if;
469
470          scaler_data_o(7 downto 0)     <= std_logic_vector(unsigned(A));
471          scaler_data_o(15 downto 8)    <= std_logic_vector(unsigned(B));
472          scaler_data_o(23 downto 16)   <= std_logic_vector(unsigned(C));
473
474          -- Handle reset
475          if sreset_i = '1' then
476             x_count <= 1; -- Needs to be 1 because of dx/dy algorithm
477             y_count <= 1; -- Needs to be 1 because of dx/dy algorithm
478          end if;
479       end if;
480    end process p_reverse_mapping;
481
482    p_scaling_ratio : process(clk_i) is
483    begin
484       if rising_edge(clk_i) then
485          -- Calc scaling ratio
486          -- Needs to be inside clocked process to become registers for fixed point DSP implementation
487          rx_height         <= to_ufixed(g_rx_video_height, rx_height);
488          tx_height         <= to_ufixed(g_tx_video_height, tx_height);
489          rx_height_reg     <= rx_height;
490          tx_height_reg     <= tx_height;
491          scaling_ratio     <= resize(rx_height_reg/tx_height_reg, scaling_ratio'high, scaling_ratio'low);
492          scaling_ratio_reg <= scaling_ratio;
493       end if;
494    end process p_scaling_ratio;
495
496 end scaler_arc;
```

A.8 Scaler Controller

```vhdl
-------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-01-21
-- Version: 0.1
-------------------------------------------------------
-- Description: Controller for the scaler design.
--              Decodes the package received and
--              generates a new control packet.
--              Hardcoded for 80-bit wide data bus,
--              and 10-bit per pixel.
-------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity scaler_controller is
   generic (
      g_data_width              : natural;
      g_empty_width             : natural;
      g_tx_video_width          : natural;
      g_tx_video_height         : natural;
      g_tx_video_scaling_method : natural
   );
   port (
      clk_i            : in  std_logic;
      sreset_i         : in  std_logic;
      -- scaler -> scaler_controller
      ctrl_startofpacket_i   : in  std_logic;
      ctrl_endofpacket_i     : in  std_logic;
      ctrl_data_i            : in  std_logic_vector(g_data_width-1 downto 0);
      ctrl_empty_i           : in  std_logic_vector(g_empty_width-1 downto 0);
      ctrl_valid_i           : in  std_logic;
      ctrl_ready_o           : out std_logic := '0';

      -- scaler_controller -> scaler
      ctrl_startofpacket_o   : out std_logic := '0';
      ctrl_endofpacket_o     : out std_logic := '0';
      ctrl_data_o            : out std_logic_vector(g_data_width-1 downto 0) := (others => '0');
      ctrl_empty_o           : out std_logic_vector(g_empty_width-1 downto 0) := (others => '0');
      ctrl_valid_o           : out std_logic := '0';
      ctrl_ready_i           : in  std_logic;

      -- Config
      rx_video_width_o           : out std_logic_vector(15 downto 0);
      rx_video_height_o          : out std_logic_vector(15 downto 0)
      );
   end entity scaler_controller;

architecture scaler_controller_arc of scaler_controller is
   type t_packet_type is (s_idle, s_video_data, s_control_packet);
   signal state   : t_packet_type := s_idle;

begin
   -- Asseart ready out
   ctrl_ready_o <= ctrl_ready_i or not ctrl_valid_o;

   p_fsm : process(clk_i) is
      variable v_tx_video_width : std_logic_vector(15 downto 0);
      variable v_tx_video_height : std_logic_vector(15 downto 0);
   begin
      if rising_edge(clk_i) then
         if ctrl_ready_i = '1' then
            ctrl_valid_o <= '0';
         end if;


         case state is
            when s_idle =>
               if ctrl_ready_o = '1' and ctrl_valid_i = '1' then
                  if ctrl_startofpacket_i = '1' and ctrl_data_i(3 downto 0) = "0000" then
                     -- Send startofpacket and video packet identifier to output
                     ctrl_data_o  <= (3 downto 0 => '0', others => '1'); -- Using others => 1 for easy identification
                         ↪   in modelsim
                     ctrl_valid_o   <= '1';
                     ctrl_startofpacket_o <= '1';

                     -- Next state
                     state <= s_video_data;
                  elsif ctrl_startofpacket_i = '1' and ctrl_data_i(3 downto 0) = "1111" then
                     -- Send startofpacket and ctrl pkg identifier to output
                     ctrl_data_o  <= (3 downto 0 => '1', others => '0');
                     ctrl_valid_o   <= '1';
                     ctrl_startofpacket_o <= '1';
```

```
86
87                      -- Next state
88                      state <= s_control_packet;
89                   end if;
90
91                   -- Reset endofpacket
92                   ctrl_endofpacket_o <= '0';
93                end if;
94
95
96             when s_video_data =>
97                if ctrl_ready_o = '1' and ctrl_valid_i = '1' then
98                   if ctrl_endofpacket_i = '1' then
99                      ctrl_endofpacket_o <= '1';
100
101                     -- Next state
102                     state <= s_idle;
103                  else
104                     -- Next state
105                     state <= s_video_data;
106                  end if;
107                  ctrl_data_o  <= ctrl_data_i;
108                  ctrl_valid_o   <= '1';
109                  ctrl_startofpacket_o <= '0';
110               end if;
111
112
113            when s_control_packet =>
114               if ctrl_ready_o = '1' and ctrl_valid_i = '1' then
115                  if g_data_width = 80 then
116                     -- Decode input video resolution
117                     rx_video_width_o(3 downto 0)      <= ctrl_data_i(33 downto 30);
118                     rx_video_width_o(7 downto 4)      <= ctrl_data_i(23 downto 20);
119                     rx_video_width_o(11 downto 8)     <= ctrl_data_i(13 downto 10);
120                     rx_video_width_o(15 downto 12)    <= ctrl_data_i(3 downto 0);
121                     rx_video_height_o(3 downto 0)     <= ctrl_data_i(73 downto 70);
122                     rx_video_height_o(7 downto 4)     <= ctrl_data_i(63 downto 60);
123                     rx_video_height_o(11 downto 8)    <= ctrl_data_i(53 downto 50);
124                     rx_video_height_o(15 downto 12)   <= ctrl_data_i(43 downto 40);
125
126                     -- Set output to slv format
127                     v_tx_video_width  := std_logic_vector(to_unsigned(g_tx_video_width, v_tx_video_width'length));
128                     v_tx_video_height := std_logic_vector(to_unsigned(g_tx_video_height, v_tx_video_height'length));
129
130                     -- Send output resolution and endofpacket
131                     ctrl_data_o(3 downto 0)   <= v_tx_video_width(15 downto 12);
132                     ctrl_data_o(13 downto 10) <= v_tx_video_width(11 downto 8);
133                     ctrl_data_o(23 downto 20) <= v_tx_video_width(7 downto 4);
134                     ctrl_data_o(33 downto 30) <= v_tx_video_width(3 downto 0);
135                     ctrl_data_o(43 downto 40) <= v_tx_video_height(15 downto 12);
136                     ctrl_data_o(53 downto 50) <= v_tx_video_height(11 downto 8);
137                     ctrl_data_o(63 downto 60) <= v_tx_video_height(7 downto 4);
138                     ctrl_data_o(73 downto 70) <= v_tx_video_height(3 downto 0);
139
140                     ctrl_valid_o   <= '1';
141                     ctrl_startofpacket_o <= '0';
142                     ctrl_endofpacket_o <= '1';
143                  end if;
144                  -- Next state
145                  state <= s_idle;
146               end if;
147
148         end case;
149
150         if sreset_i = '1' then
151            ctrl_valid_o <= '0';
152            state <= s_idle;
153         end if;
154      end if;
155   end process p_fsm;
156 end scaler_controller_arc;
```

A.9 Scaler Top Level

```
1  -------------------------------------------------------
2  -- Project: FPGA video scaler
3  -- Author: Thomas Stenseth
4  -- Date: 2019-01-21
5  -- Version: 0.1
6  -------------------------------------------------------
7  -- Description: Top-level of scaler
8  -------------------------------------------------------
9
10
11 library ieee;
12 use ieee.std_logic_1164.all;
13 use ieee.numeric_std.all;
14
15
16 entity scaler_wrapper is
17    generic (
18       g_data_width                : natural;
19       g_empty_width               : natural;
20       g_fifo_data_width           : natural;
21       g_fifo_data_depth           : natural;
22       g_tx_video_width            : natural;
23       g_tx_video_height           : natural;
24       g_tx_video_scaling_method   : natural
25    );
26    port (
27       clk_i       : in  std_logic;
28       sreset_i    : in  std_logic;
29       -- To scaler
30       startofpacket_i   : in  std_logic;
31       endofpacket_i     : in  std_logic;
32       data_i            : in  std_logic_vector(g_data_width-1 downto 0);
33       empty_i           : in  std_logic_vector(g_empty_width-1 downto 0);
34       valid_i           : in  std_logic;
35       ready_o           : out std_logic := '0';
36
37       -- From scaler
38       startofpacket_o   : out std_logic := '0';
39       endofpacket_o     : out std_logic := '0';
40       data_o            : out std_logic_vector(g_data_width-1 downto 0) := (others => '0');
41       empty_o           : out std_logic_vector(g_empty_width-1 downto 0)  := (others => '0');
42       valid_o           : out std_logic := '0';
43       ready_i           : in  std_logic
44    );
45 end entity scaler_wrapper;
46
47 architecture scaler_wrapper_arc of scaler_wrapper is
48    ---- FIFO constants
49    --constant c_data_range_fifo    : natural := g_data_width;
50    --constant c_empty_range_fifo   : natural := g_data_width + g_empty_width;
51    --constant c_sop_range_fifo     : natural := g_data_width + g_empty_width + 1;
52    --constant c_eop_range_fifo     : natural := g_data_width + g_empty_width + 2;
53
54    -- Controller
55    signal ctrl_ready_i  : std_logic;
56    signal ctrl_ready_o  : std_logic;
57    signal ctrl_valid_i  : std_logic;
58    signal ctrl_valid_o  : std_logic;
59    signal ctrl_data_i   : std_logic_vector(g_data_width-1 downto 0);
60    signal ctrl_data_o   : std_logic_vector(g_data_width-1 downto 0);
61
62    signal rx_video_width_o           : std_logic_vector(15 downto 0);
63    signal rx_video_height_o          : std_logic_vector(15 downto 0);
64
65    -- Scaler
66    signal scaler_ready_i  : std_logic;
67    signal scaler_ready_o  : std_logic;
68    signal scaler_valid_i  : std_logic;
69    signal scaler_valid_o  : std_logic;
70    signal scaler_data_i   : std_logic_vector(g_data_width-1 downto 0);
71    signal scaler_data_o   : std_logic_vector(g_data_width-1 downto 0);
72
73    ---- Input FIFO
74    --signal fifo_in_wr_en_i        : std_logic;
75    --signal fifo_in_rd_en_i        : std_logic;
76    --signal fifo_in_data_i         : std_logic_vector(g_fifo_data_width-1 downto 0);
77    --signal fifo_in_full_o         : std_logic;
78    --signal fifo_in_almostfull_o   : std_logic;
79    --signal fifo_in_empty_o        : std_logic;
80    --signal fifo_in_data_o         : std_logic_vector(g_fifo_data_width-1 downto 0);
81    --signal fifo_in_data_reg       : std_logic_vector(g_fifo_data_width-1 downto 0);
82
83 begin
84    ------------------------------------------------
85    -- CONTROLLER
86    ------------------------------------------------
```

```
87
88      scaler_controller : entity work.scaler_controller
89      generic map(
90         g_data_width      => g_data_width,
91         g_empty_width     => g_empty_width,
92         g_tx_video_width  => g_tx_video_width,
93         g_tx_video_height => g_tx_video_height,
94         g_tx_video_scaling_method => g_tx_video_scaling_method
95      )
96
97      port map(
98         clk_i             => clk_i,
99         sreset_i          => sreset_i,
100        -- To scaler_controller
101        --startofpacket_i   => fifo_in_data_o(c_sop_range_fifo-1),
102        --endofpacket_i     => fifo_in_data_o(c_eop_range_fifo-1),
103        --data_i            => fifo_in_data_o(c_data_range_fifo-1 downto 0),
104        --empty_i           => fifo_in_data_o(c_empty_range_fifo-1 downto c_data_range_fifo),
105        ctrl_startofpacket_i   => startofpacket_i,
106        ctrl_endofpacket_i     => endofpacket_i,
107        ctrl_data_i            => ctrl_data_i,
108        ctrl_empty_i           => empty_i,
109        ctrl_valid_i           => ctrl_valid_i,
110        ctrl_ready_o           => ctrl_ready_o,
111
112        -- From scaler_controller
113        ctrl_startofpacket_o   => startofpacket_o,
114        ctrl_endofpacket_o     => endofpacket_o,
115        ctrl_data_o            => ctrl_data_o,
116        ctrl_empty_o           => empty_o,
117        ctrl_valid_o           => ctrl_valid_o,
118        ctrl_ready_i           => ctrl_ready_i,
119
120        -- Config
121        rx_video_width_o           => rx_video_width_o,
122        rx_video_height_o          => rx_video_height_o
123
124        ---- Input FIFO
125        --fifo_in_wr_en_i          => fifo_in_wr_en_i,
126        --fifo_in_rd_en_i          => fifo_in_rd_en_i,
127        --fifo_in_full_o           => fifo_in_full_o,
128        --fifo_in_almostfull_o     => fifo_in_almostfull_o,
129        --fifo_in_empty_o          => fifo_in_empty_o
130     );
131
132     -- Test without scaler
133     ctrl_ready_i   <= ready_i;
134     ctrl_valid_i   <= valid_i;
135     ctrl_data_i    <= data_i;
136     ready_o        <= ctrl_ready_o;
137     valid_o        <= ctrl_valid_o;
138     data_o         <= ctrl_data_o;
139
140
141     ------------------------------------------------
142     -- SCALER
143     ------------------------------------------------
144
145     --scaler : entity work.scaler
146     --generic map(
147     --      g_data_width => g_data_width,
148     --      g_tx_video_width => g_tx_video_width,
149     --      g_tx_video_height => g_tx_video_height
150     --   )
151     --port map(
152     --   clk_i => clk_i,
153     --   sreset_i => sreset_i,
154
155     --   scaler_data_i  => scaler_data_i,
156     --   scaler_valid_i => scaler_valid_i,
157     --   scaler_ready_o => scaler_ready_o,
158
159     --   scaler_data_o  => scaler_data_o,
160     --   scaler_valid_o => scaler_valid_o,
161     --   scaler_ready_i => scaler_ready_i
162     --);
163
164     --scaler_ready_i    <= ready_i;
165     --valid_o           <= scaler_valid_o;
166     --data_o            <= scaler_data_o;
167
168     --ctrl_ready_i      <= scaler_ready_o;
169     --scaler_valid_i    <= ctrl_valid_o;
170     --scaler_data_i     <= ctrl_data_o;
171
172     --ready_o           <= ctrl_ready_o;
173     --ctrl_valid_i      <= valid_i;
174     --ctrl_data_i       <= data_i;
175
176
```

```
177     ------------------------------------------------
178     -- FIFO
179     ------------------------------------------------
180
181     --fifo_in : entity work.fifo_generic
182     --generic map (
183     --   g_width        => g_fifo_data_width,
184     --   g_depth        => g_fifo_data_depth,
185     --   g_ramstyle     => "M20K",
186     --   g_output_reg   => true
187     --)
188     --port map(
189     --   clk_i          => clk_i,
190     --   sreset_i       => sreset_i,
191     --   wr_en_i        => fifo_in_wr_en_i,
192     --   rd_en_i        => fifo_in_rd_en_i,
193     --   data_i         => fifo_in_data_i,
194     --   full_o         => fifo_in_full_o,
195     --   almostfull_o   => fifo_in_almostfull_o,
196     --   empty_o        => fifo_in_empty_o,
197     --   data_o         => fifo_in_data_o
198     --);
199
200     --p_fill_fifo_in : process(clk_i) is
201     --begin
202     --   if rising_edge(clk_i) then
203     --      -- Assert ready out as long as there is room in FIFO
204     --      if fifo_in_almostfull_o = '1' or fifo_in_full_o = '1' then
205     --         scaler_ready_o <= '0';
206     --      else
207     --         scaler_ready_o <= '1';
208     --      end if;
209
210     --      -- Write to FIFO on valid_i if FIFO is not full
211     --      if scaler_valid_i = '1' and fifo_in_full_o = '0' then
212     --         fifo_in_wr_en_i <= '1';
213     --      else
214     --         fifo_in_wr_en_i <= '0';
215     --      end if;
216
217     --      -- Empty FIFO when controller is ready and FIFO is not empty
218     --      if ctrl_ready_o = '1' and fifo_in_empty_o = '0' then
219     --         fifo_in_rd_en_i   <= '1';
220     --         ctrl_valid_i      <= '1';
221     --      else
222     --         fifo_in_rd_en_i   <= '0';
223     --         ctrl_valid_i      <= '0';
224     --      end if;
225     --   end if; -- rising_edge(clk_i)
226     --end process p_fill_fifo_in;
227
228     ---- NOT CORRECT!!!!!
229     --p_fifo_in : process(clk_i) is
230     --begin
231     --   if rising_edge(clk_i) then
232     --      if fifo_in_rd_en_i = '1' then
233     --         ctrl_valid_i <= '1';
234     --      end if;
235     --   end if;
236     --end process p_fifo_in;
237
238     ---- Read/write to fifo
239     --scaler_ready_o <= not(fifo_in_almostfull_o);
240     --fifo_in_wr_en_i <= '1' when scaler_valid_i = '1' and fifo_in_full_o = '0' else '0';
241     --fifo_in_rd_en_i <= '1' when (ctrl_ready_o = '1' and fifo_in_empty_o = '0') else '0';
242
243
244     ---- Map input data signals to input FIFO
245     --fifo_in_data_i(c_data_range_fifo-1 downto 0)                   <= scaler_data_i;
246     --fifo_in_data_i(c_empty_range_fifo-1 downto c_data_range_fifo)  <= scaler_empty_i;
247     --fifo_in_data_i(c_sop_range_fifo-1)                             <= scaler_sop_i;
248     --fifo_in_data_i(c_eop_range_fifo-1)                             <= scaler_eop_i;
249
250     ---- FIFO output
251     --ctrl_ready_i <= '1';
252
253  end scaler_wrapper_arc;
```

Appendix B

VHDL Testbenches

B.1 FIFO Testbench

```
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-03-11
-- Version: 0.1
--------------------------------------------------------
-- Description: Testbench for FIFO
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;


-- Test bench entity
entity tb_fifo_generic is
end tb_fifo_generic;

-- Test bench architecture
architecture tb_fifo_generic_arc of tb_fifo_generic is
   constant C_SCOPE         : string  := C_TB_SCOPE_DEFAULT;
   constant C_CLK_PERIOD    : time := 10 ns; -- 100 MHz

   -- Width and depth of FIFO
   constant C_WIDTH      : natural   := 20;
   constant C_DEPTH      : natural   := 10;

   -- Clk, sreset
   signal clk_i          : std_logic;
   signal sreset_i       : std_logic;
   -- Write to fifo
   signal data_i         : std_logic_vector(C_WIDTH-1 downto 0) := (others => '0');
   signal wr_en_i        : std_logic;
   signal full_o         : std_logic;
   signal almostfull_o   : std_logic;
   -- Read from fifo
   signal data_o         : std_logic_vector(C_WIDTH-1 downto 0);
   signal rd_en_i        : std_logic;
   signal empty_o        : std_logic;

begin
   -----------------------------------------------------------------------------
   -- Instantiate the concurrent procedure that initializes UVVM
   -----------------------------------------------------------------------------
```

```
53      i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;
54
55
56      --------------------------------------------------------------------------------
57      -- Instantiate DUT
58      --------------------------------------------------------------------------------
59      i_fifo: entity work.fifo_generic
60      generic map(
61         g_width     => C_WIDTH,
62         g_depth     => C_DEPTH
63      )
64      port map(
65         clk_i          => clk_i,
66         sreset_i       => sreset_i,
67         data_i         => data_i,
68         wr_en_i        => wr_en_i,
69         full_o         => full_o,
70         almostfull_o   => almostfull_o,
71         data_o         => data_o,
72         rd_en_i        => rd_en_i,
73         empty_o        => empty_o
74      );
75
76
77      --------------------------------------------------------------------------------
78      -- Reset process
79      --------------------------------------------------------------------------------
80      -- Toggle the reset after 5 clock periods
81      p_sreset: sreset_i <= '1', '0' after 5 *C_CLK_PERIOD;
82
83
84      --------------------------------------------------------------------------------
85      -- Clock process
86      --------------------------------------------------------------------------------
87      p_clk: process
88      begin
89         clk_i <= '0', '1' after C_CLK_PERIOD / 2;
90         wait for C_CLK_PERIOD;
91      end process;
92
93      --------------------------------------------------------------------------------
94      -- Data_i generate random data process
95      --------------------------------------------------------------------------------
96      p_data_i : process(clk_i)
97      begin
98         if rising_edge(clk_i) then
99            data_i <= random(C_WIDTH);
100        end if;
101     end process;
102
103     ------------------------------------------------
104     -- PROCESS: p_main
105     ------------------------------------------------
106     p_main: process
107     begin
108        -- Wait for UVVM to finish initialization
109        await_uvvm_initialization(VOID);
110
111        -- Print the configuration to the log
112        report_global_ctrl(VOID);
113        report_msg_id_panel(VOID);
114
115        --------------------------------------------------------------------------------
116        -- Enable log message
117        --------------------------------------------------------------------------------
118        enable_log_msg(ALL_MESSAGES);
119
120        log(ID_LOG_HDR, "Starting simulation of FIFO", C_SCOPE);
121        log("Wait 10 clock period for reset to be turned off");
122        wait for (10 * C_CLK_PERIOD);
123
124        --------------------------------------------------------------------------------
125        -- Test FIFO
126        --------------------------------------------------------------------------------
127        wr_en_i <= '0';
128        rd_en_i <= '0';
129        wait until rising_edge(clk_i);
130
131        -- Fill FIFO
132        for i in 1 to C_DEPTH*2 loop
133           wr_en_i <= '1';
134           rd_en_i <= '0';
135           wait until rising_edge(clk_i);
136        end loop;
137
138        -- Empty FIFO
139        for i in 1 to C_DEPTH*2 loop
140           wr_en_i <= '0';
141           rd_en_i <= '1';
142           wait until rising_edge(clk_i);
```

```vhdl
143        end loop;
144
145        -- Idle
146        for i in 1 to C_DEPTH*2 loop
147           wr_en_i <= '0';
148           rd_en_i <= '0';
149           wait until rising_edge(clk_i);
150        end loop;
151
152        -- Stream through empty FIFO
153        for i in 1 to C_DEPTH*2 loop
154           wr_en_i <= '1';
155           rd_en_i <= '1';
156           wait until rising_edge(clk_i);
157        end loop;
158
159        -- Empty FIFO
160        for i in 1 to C_DEPTH*2 loop
161           wr_en_i <= '0';
162           rd_en_i <= '1';
163           wait until rising_edge(clk_i);
164        end loop;
165
166        -- Idle
167        for i in 1 to C_DEPTH*2 loop
168           wr_en_i <= '0';
169           rd_en_i <= '0';
170           wait until rising_edge(clk_i);
171        end loop;
172
173
174        ----------------------------------------------------------------------------
175        -- Ending the simulation
176        ----------------------------------------------------------------------------
177        wait for 1000 ns;             -- to allow some time for completion
178        report_alert_counters(FINAL); -- Report final counters and print conclusion for simulation (Success/Fail)
179        log(ID_LOG_HDR, "SIMULATION COMPLETED", C_SCOPE);
180
181        -- Finish the simulation
182        std.env.stop;
183        wait;  -- to stop completely
184     end process p_main;
185
186
187  end tb_fifo_generic_arc;
```

B.2 Simple Dual-Port RAM Testbench

```vhdl
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-03-11
-- Version: 0.1
--------------------------------------------------------
-- Description: Testbench for simple dual-port RAM
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;


-- Test bench entity
entity tb_simple_dpram is
end tb_simple_dpram;

-- Test bench architecture
architecture tb_simple_dpram_arc of tb_simple_dpram is
   constant C_SCOPE        : string  := C_TB_SCOPE_DEFAULT;
   constant C_CLK_PERIOD   : time := 10 ns; -- 100 MHz

   -- RAM width and depth
   constant C_RAM_WIDTH    : natural := 20;
   constant C_RAM_DEPTH    : natural := 10;

   signal clk_i      : std_logic;
   signal data_i     : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');
   signal wr_addr_i  : integer := 0;
   signal wr_en_i    : std_logic := '0';
   signal rd_addr_i  : integer := 0;
   signal data_o     : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');

begin
   -----------------------------------------------------------------------------
   -- Instantiate the concurrent procedure that initializes UVVM
   -----------------------------------------------------------------------------
   i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;


   -----------------------------------------------------------------------------
   -- Instantiate DUT
   -----------------------------------------------------------------------------
   i_simple_dpram: entity work.simple_dpram
   generic map(
      g_ram_width     => C_RAM_WIDTH,
      g_ram_depth     => C_RAM_DEPTH
   )
   port map(
      clk_i       => clk_i,
      data_i      => data_i,
      wr_addr_i   => wr_addr_i,
      wr_en_i     => wr_en_i,
      rd_addr_i   => rd_addr_i,
      data_o      => data_o
   );


   ------------------------------------------------
   -- PROCESS: p_main
   ------------------------------------------------
   p_main: process
      variable v_writeaddr : natural := 0;
      variable v_readaddr  : natural := 0;
   begin
      -- Wait for UVVM to finish initialization
      await_uvvm_initialization(VOID);

      -- Print the configuration to the log
      report_global_ctrl(VOID);
      report_msg_id_panel(VOID);

      -----------------------------------------------------------------------------
      -- Enable log message
      -----------------------------------------------------------------------------
      enable_log_msg(ALL_MESSAGES);

```

```
87          log(ID_LOG_HDR, "Starting simulation of FIFO", C_SCOPE);
88          log("Wait 10 clock period for reset to be turned off");
89          wait for (10 * C_CLK_PERIOD);
90
91          ----------------------------------------------------------------------------
92          -- Test simple dual-port RAM
93          ----------------------------------------------------------------------------
94          wait until rising_edge(clk_i);
95
96          -- Write random data to RAM
97          wr_en_i <= '1';
98          for i in 1 to C_RAM_DEPTH loop
99             data_i      <= random(C_RAM_WIDTH);
100            wr_addr_i   <= v_writeaddr;
101            v_writeaddr   := v_writeaddr + 1 when (v_writeaddr < C_RAM_DEPTH-1) else 0;
102            wait until rising_edge(clk_i);
103         end loop;
104         wr_en_i <= '0';
105
106         -- Read from RAM
107         for i in 1 to C_RAM_DEPTH loop
108            rd_addr_i   <= v_readaddr;
109            v_readaddr    := v_readaddr + 1 when (v_readaddr < C_RAM_DEPTH-1) else 0;
110            wait until rising_edge(clk_i);
111         end loop;
112
113         -- Random fill up RAM
114         wr_en_i <= '1';
115         for i in 1 to C_RAM_DEPTH loop
116            data_i      <= random(C_RAM_WIDTH);
117            wr_addr_i   <= random(0,C_RAM_DEPTH-1);
118            wait until rising_edge(clk_i);
119         end loop;
120         wr_en_i <= '0';
121
122         -- Random read RAM
123         for i in 1 to C_RAM_DEPTH loop
124            rd_addr_i   <= random(0,C_RAM_DEPTH-1);
125            wait until rising_edge(clk_i);
126         end loop;
127
128         -- Concurrent read and write form random addresses
129         wr_en_i <= '1';
130         for i in 1 to 5*C_RAM_DEPTH loop
131            data_i      <= random(C_RAM_WIDTH);
132            wr_addr_i   <= random(0,C_RAM_DEPTH-1);
133            rd_addr_i   <= random(0,C_RAM_DEPTH-1);
134            wait until rising_edge(clk_i);
135         end loop;
136         wr_en_i <= '0';
137
138
139         ----------------------------------------------------------------------------
140         -- Ending the simulation
141         ----------------------------------------------------------------------------
142         wait for 1000 ns;              -- to allow some time for completion
143         report_alert_counters(FINAL); -- Report final counters and print conclusion for simulation (Success/Fail)
144         log(ID_LOG_HDR, "SIMULATION COMPLETED", C_SCOPE);
145
146         -- Finish the simulation
147         std.env.stop;
148         wait;  -- to stop completely
149      end process p_main;
150
151
152      ----------------------------------------------------------------------------
153      -- Clock process
154      ----------------------------------------------------------------------------
155      p_clk: process
156      begin
157         clk_i <= '0', '1' after C_CLK_PERIOD / 2;
158         wait for C_CLK_PERIOD;
159      end process;
160
161   end tb_simple_dpram_arc;
```

B.3 Multiport RAM Testbench

```vhdl
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-03-11
-- Version: 0.1
--------------------------------------------------------
-- Description: Testbench for multiport RAM
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;


-- Test bench entity
entity tb_multiport_ram is
end tb_multiport_ram;

-- Test bench architecture
architecture tb_multiport_ram_arc of tb_multiport_ram is
   constant C_SCOPE         : string  := C_TB_SCOPE_DEFAULT;
   constant C_CLK_PERIOD    : time := 10 ns; -- 100 MHz

   -- RAM width and depth
   constant C_RAM_WIDTH     : natural := 30;
   constant C_RAM_DEPTH     : natural := 10;

   signal clk_i          : std_logic;
   signal data_i         : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');
   signal wr_addr_i      : integer := 0;
   signal wr_en_i        : std_logic := '0';
   signal rd_addr_a_i    : integer := 0;
   signal rd_addr_b_i    : integer := 0;
   signal rd_addr_c_i    : integer := 0;
   signal rd_addr_d_i    : integer := 0;
   signal data_a_o       : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');
   signal data_b_o       : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');
   signal data_c_o       : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');
   signal data_d_o       : std_logic_vector(C_RAM_WIDTH-1 downto 0) := (others =>'0');

begin
   -----------------------------------------------------------------------------
   -- Instantiate the concurrent procedure that initializes UVVM
   -----------------------------------------------------------------------------
   i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;


   -----------------------------------------------------------------------------
   -- Instantiate DUT
   -----------------------------------------------------------------------------
   i_multiport_ram: entity work.multiport_ram
   generic map(
      g_ram_width     => C_RAM_WIDTH,
      g_ram_depth     => C_RAM_DEPTH
   )
   port map(
      clk_i       => clk_i,
      data_i      => data_i,
      wr_addr_i   => wr_addr_i,
      wr_en_i     => wr_en_i,
      data_a_o    => data_a_o,
      data_b_o    => data_b_o,
      data_c_o    => data_c_o,
      data_d_o    => data_d_o,
      rd_addr_a_i => rd_addr_a_i,
      rd_addr_b_i => rd_addr_b_i,
      rd_addr_c_i => rd_addr_c_i,
      rd_addr_d_i => rd_addr_d_i
   );


   ------------------------------------------------
   -- PROCESS: p_main
   ------------------------------------------------
   p_main: process
      variable v_writeaddr    : natural := 0;
      variable v_readaddr_a   : natural := 0;
      variable v_readaddr_b   : natural := 0;
```

```
87        variable v_readaddr_c    : natural := 0;
88        variable v_readaddr_d    : natural := 0;
89     begin
90        -- Wait for UVVM to finish initialization
91        await_uvvm_initialization(VOID);
92
93        -- Print the configuration to the log
94        report_global_ctrl(VOID);
95        report_msg_id_panel(VOID);
96
97        --------------------------------------------------------------------------
98        -- Enable log message
99        --------------------------------------------------------------------------
100       enable_log_msg(ALL_MESSAGES);
101
102       log(ID_LOG_HDR, "Starting simulation of FIFO", C_SCOPE);
103       log("Wait 10 clock period for reset to be turned off");
104       wait for (10 * C_CLK_PERIOD);
105
106       --------------------------------------------------------------------------
107       -- Test simple dual-port RAM
108       --------------------------------------------------------------------------
109       wait until rising_edge(clk_i);
110
111       -- Write random data to RAM
112       wr_en_i <= '1';
113       for i in 1 to C_RAM_DEPTH loop
114          data_i      <= random(C_RAM_WIDTH);
115          wr_addr_i   <= v_writeaddr;
116          v_writeaddr := v_writeaddr + 1 when (v_writeaddr < C_RAM_DEPTH-1) else 0;
117          wait until rising_edge(clk_i);
118       end loop;
119       wr_en_i <= '0';
120
121       -- Read from RAM
122       for i in 1 to C_RAM_DEPTH loop
123          rd_addr_a_i    <= v_readaddr_a;
124          rd_addr_b_i    <= v_readaddr_b;
125          rd_addr_c_i    <= v_readaddr_c;
126          rd_addr_d_i    <= v_readaddr_d;
127          v_readaddr_a   := v_readaddr_a + 1 when (v_readaddr_a < C_RAM_DEPTH-1) else 0;
128          v_readaddr_b   := v_readaddr_b + 1 when (v_readaddr_b < C_RAM_DEPTH-1) else 0;
129          v_readaddr_c   := v_readaddr_c + 1 when (v_readaddr_c < C_RAM_DEPTH-1) else 0;
130          v_readaddr_d   := v_readaddr_d + 1 when (v_readaddr_d < C_RAM_DEPTH-1) else 0;
131          wait until rising_edge(clk_i);
132       end loop;
133
134       -- Random fill up RAM
135       wr_en_i <= '1';
136       for i in 1 to C_RAM_DEPTH loop
137          data_i      <= random(C_RAM_WIDTH);
138          wr_addr_i   <= random(0,C_RAM_DEPTH-1);
139          wait until rising_edge(clk_i);
140       end loop;
141       wr_en_i <= '0';
142
143       -- Random read RAM
144       for i in 1 to C_RAM_DEPTH loop
145          rd_addr_a_i <= random(0,C_RAM_DEPTH-1);
146          rd_addr_b_i <= random(0,C_RAM_DEPTH-1);
147          rd_addr_c_i <= random(0,C_RAM_DEPTH-1);
148          rd_addr_d_i <= random(0,C_RAM_DEPTH-1);
149          wait until rising_edge(clk_i);
150       end loop;
151
152       -- Concurrent read and write form random addresses
153       wr_en_i <= '1';
154       for i in 1 to 5*C_RAM_DEPTH loop
155          data_i      <= random(C_RAM_WIDTH);
156          wr_addr_i   <= random(0,C_RAM_DEPTH-1);
157          rd_addr_a_i <= random(0,C_RAM_DEPTH-1);
158          rd_addr_b_i <= random(0,C_RAM_DEPTH-1);
159          rd_addr_c_i <= random(0,C_RAM_DEPTH-1);
160          rd_addr_d_i <= random(0,C_RAM_DEPTH-1);
161          wait until rising_edge(clk_i);
162       end loop;
163       wr_en_i <= '0';
164
165
166       --------------------------------------------------------------------------
167       -- Ending the simulation
168       --------------------------------------------------------------------------
169       wait for 1000 ns;             -- to allow some time for completion
170       report_alert_counters(FINAL); -- Report final counters and print conclusion for simulation (Success/Fail)
171       log(ID_LOG_HDR, "SIMULATION COMPLETED", C_SCOPE);
172
173       -- Finish the simulation
174       std.env.stop;
175       wait;  -- to stop completely
176    end process p_main;
```

```
177
178
179     ---------------------------------------------------------------------------
180     -- Clock process
181     ---------------------------------------------------------------------------
182     p_clk: process
183     begin
184        clk_i <= '0', '1' after C_CLK_PERIOD / 2;
185        wait for C_CLK_PERIOD;
186     end process;
187
188  end tb_multiport_ram_arc;
```

B.4 Scaler Algorithm Testbench With File IO

```vhdl
--------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-04-14
-- Version: 0.1
--------------------------------------------------------
-- Description: Testbench for scaler algorithm
--------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;


-- Test bench entity
entity tb_scaler is
end tb_scaler;

-- Test bench architecture
architecture tb_scaler_arc of tb_scaler is
   constant C_SCOPE        : string := C_TB_SCOPE_DEFAULT;
   constant C_CLK_PERIOD   : time   := 10 ns; -- 100 MHz

   -- Avalon-ST bus widths
   constant C_DATA_WIDTH      : natural := 24;
   constant C_BITS_PIXEL      : natural := 8;

   constant C_RX_VIDEO_WIDTH  : natural := 640;
   constant C_RX_VIDEO_HEIGHT : natural := 360;
   constant C_TX_VIDEO_WIDTH  : natural := 1920;
   constant C_TX_VIDEO_HEIGHT : natural := 1080;

   -- File I/O
   constant C_IMAGE        : string := "lionking";
   constant C_SCALING      : string := "nearest";
   constant C_INPUT_FILE   : string := "../../data/orig/" & C_IMAGE & "/" & C_IMAGE & "_ycbcr444_" &
      ↪    to_string(C_BITS_PIXEL) & "bit_" & to_string(C_RX_VIDEO_HEIGHT) & ".bin";
   constant C_OUTPUT_FILE  : string := "../../data/vhdl_out/" & C_IMAGE & "/" & C_IMAGE & "_" & C_SCALING & "_" &
      ↪    to_string(C_RX_VIDEO_HEIGHT) & "_to_" & to_string(C_TX_VIDEO_HEIGHT) & ".bin";
   file     file_input     : text;
   file     file_output    : text;

   -- DSP interface and general control signals
   signal clk_i               : std_logic := '0';
   signal sreset_i            : std_logic := '0';

  -- DUT scaler inputs
   signal startofpacket_i     : std_logic := '0';
   signal endofpacket_i       : std_logic := '0';
   signal data_i              : std_logic_vector(C_DATA_WIDTH-1 downto 0) := (others => '0');
   signal valid_i             : std_logic := '0';
   signal ready_i             : std_logic := '0';

   -- DUT scaler outputs
   signal startofpacket_o     : std_logic := '0';
   signal endofpacket_o       : std_logic := '0';
   signal data_o              : std_logic_vector(C_DATA_WIDTH-1 downto 0) := (others => '0');
   signal valid_o             : std_logic := '0';
   signal ready_o             : std_logic := '0';

begin
   -----------------------------------------------------------------------------
   -- Instantiate the concurrent procedure that initializes UVVM
   -----------------------------------------------------------------------------
   i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;



   -----------------------------------------------------------------------------
   -- Instantiate DUT
   -----------------------------------------------------------------------------
   i_scaler: entity work.scaler
   generic map(
      g_data_width         => C_DATA_WIDTH,
      g_rx_video_width     => C_RX_VIDEO_WIDTH,
      g_rx_video_height    => C_RX_VIDEO_HEIGHT,
      g_tx_video_width     => C_TX_VIDEO_WIDTH,
```

```
85        g_tx_video_height    => C_TX_VIDEO_HEIGHT
86     )
87     port map(
88        clk_i             => clk_i,
89        sreset_i          => sreset_i,
90
91        scaler_startofpacket_i  => startofpacket_i,
92        scaler_endofpacket_i    => endofpacket_i,
93        scaler_data_i           => data_i,
94        scaler_valid_i          => valid_i,
95        scaler_ready_o          => ready_o,
96
97        scaler_startofpacket_o  => startofpacket_o,
98        scaler_endofpacket_o    => endofpacket_o,
99        scaler_data_o           => data_o,
100       scaler_valid_o          => valid_o,
101       scaler_ready_i          => ready_i
102    );
103
104
105    ------------------------------------------------
106    -- PROCESS: p_main
107    ------------------------------------------------
108    p_main: process
109       variable v_input_line      : line;
110       variable v_data_slv        : std_logic_vector(C_DATA_WIDTH-1 downto 0) := (others => '0');
111       variable v_data            : integer := 0;
112       variable v_num_test_loops  : integer := 0;
113    begin
114       -- Wait for UVVM to finish initialization
115       await_uvvm_initialization(VOID);
116
117       -- Print the configuration to the log
118       report_global_ctrl(VOID);
119       report_msg_id_panel(VOID);
120
121       ----------------------------------------------------------------------------
122       -- Enable log message
123       ----------------------------------------------------------------------------
124       enable_log_msg(ALL_MESSAGES);
125
126       log(ID_LOG_HDR, "Starting simulation of nearest", C_SCOPE);
127       log("Wait 10 clock period for reset to be turned off");
128       wait for (10 * C_CLK_PERIOD);
129       wait until rising_edge(clk_i);
130
131       ----------------------------------------------------------------------------
132       -- Test scaler
133       ----------------------------------------------------------------------------
134       v_num_test_loops := 1;
135
136       ----------------------------------------------------------------------------
137       -- Send video data control packet
138       ----------------------------------------------------------------------------
139       ready_i <= '1';
140       data_i <= (others => '0');
141       valid_i <= '1';
142       startofpacket_i <= '1';
143       wait until rising_edge(clk_i);
144       startofpacket_i <= '0';
145
146       ------------------------------------------------------------------------------
147       ---- Send known video data
148       ------------------------------------------------------------------------------
149       --for n in 1 to v_num_test_loops loop
150       --   for i in 1 to C_RX_VIDEO_WIDTH loop
151       --      v_data := (100 * i) + 1;
152       --      for j in 1 to C_RX_VIDEO_HEIGHT loop
153       --         while ready_o = '0' loop
154       --            valid_i  <= '0';
155       --            wait until rising_edge(clk_i);
156       --         end loop;
157       --         endofpacket_i   <= '1' when (i = C_RX_VIDEO_WIDTH and j = C_RX_VIDEO_HEIGHT) else '0';
158       --         data_i   <= std_logic_vector(to_unsigned(v_data, data_i'length));
159       --         valid_i  <= '1';
160       --         v_data := v_data + 1;
161       --         wait until rising_edge(clk_i);
162       --         --valid_i  <= '0';
163       --         --wait until rising_edge(clk_i);
164       --      end loop;
165       --   end loop;
166       --end loop;
167       --valid_i <= '0';
168       --endofpacket_i <= '0';
169
170       --------------------------------------------------
171       -- Read input file and send video data
172       --------------------------------------------------
173       file_open(file_input, C_INPUT_FILE, read_mode);
174
```

```
175          while not endfile(file_input) loop
176             while ready_o = '0' loop
177                valid_i  <= '0';
178                wait until rising_edge(clk_i);
179             end loop;
180
181             -- Read input data and send
182             readline(file_input, v_input_line);
183             read(v_input_line, v_data_slv);
184             data_i <= v_data_slv;
185             valid_i <= '1';
186             wait until rising_edge(clk_i);
187          end loop;
188          file_close(file_input);
189          valid_i <= '0';
190
191
192          ----------------------------------------------------------------------------
193          -- Ending the simulation
194          ----------------------------------------------------------------------------
195          wait for 10*C_CLK_PERIOD;
196          wait for C_TX_VIDEO_WIDTH*C_TX_VIDEO_HEIGHT*C_CLK_PERIOD;
197
198          report_alert_counters(FINAL); -- Report final counters and print conclusion for simulation (Success/Fail)
199          log(ID_LOG_HDR, "SIMULATION COMPLETED", C_SCOPE);
200
201          -- Finish the simulation
202          std.env.stop;
203          wait;  -- to stop completely
204       end process p_main;
205
206
207       ----------------------------------------------------------------------------
208       -- Write data to binary file
209       ----------------------------------------------------------------------------
210       p_write_data: process
211          variable v_out_line   : line;
212          variable v_data_slv   : std_logic_vector(C_DATA_WIDTH-1 downto 0) := (others => '0');
213          variable v_sop        : boolean := false;
214       begin
215          file_open(file_output, C_OUTPUT_FILE, write_mode);
216          wait until rising_edge(clk_i);
217
218          -- Wait for startofpacket_o
219          while not v_sop loop
220             wait until rising_edge(clk_i);
221             if startofpacket_o = '1' then
222                v_sop := true;
223             end if;
224          end loop;
225
226          -- Write data on eack clock as long as valid_o = '1'
227          while v_sop loop
228             if valid_o = '1' then
229                v_data_slv := data_o;
230                write(v_out_line, v_data_slv);
231                writeline(file_output, v_out_line);
232             end if;
233             wait until rising_edge(clk_i);
234          end loop;
235          file_close(file_output);
236       end process;
237
238
239       ----------------------------------------------------------------------------
240       -- Clock process
241       ----------------------------------------------------------------------------
242       p_clk: process
243       begin
244          clk_i <= '0', '1' after C_CLK_PERIOD / 2;
245          wait for C_CLK_PERIOD;
246       end process;
247
248    end tb_scaler_arc;
```

B.5 Scaler Top Level Testbench With UVVM

```vhdl
-------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-01-21
-- Version: 0.1
-------------------------------------------------------
-- Description: Testbench for scaler full design
-------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;

library vip_avalon_st;
use vip_avalon_st.vvc_methods_pkg.all;
use vip_avalon_st.td_vvc_framework_common_methods_pkg.all;


-- Test bench entity
entity tb_scaler_complete is
end entity;

-- Test bench architecture
architecture func of tb_scaler_complete is
   constant C_SCOPE              : string  := C_TB_SCOPE_DEFAULT;

   -- Clock and bit period settings
   constant C_CLK_PERIOD         : time := 10 ns;
   constant C_BIT_PERIOD         : time := 16 * C_CLK_PERIOD;

   -- Avalon-ST bus widths
   constant C_DATA_WIDTH      : natural := 80;
   constant C_EMPTY_WIDTH     : natural := 1;

   -- FIFOs
   constant C_FIFO_DATA_WIDTH : natural := C_DATA_WIDTH + C_EMPTY_WIDTH + 2;
   constant C_FIFO_DATA_DEPTH : natural := 64;

   -- File I/O
   constant C_INPUT_FILE   : string := "../../data/orig/lionking/lionking_ycbcr444_8bit_360.bin";
   constant C_EXPECT_FILE  : string := "../../data/orig/lionking/lionking_ycbcr444_8bit_360.bin";
   file     file_input     : text;
   file     file_expect    : text;

   -- Test data
   constant C_RX_VIDEO_WIDTH  : natural := 640;
   constant C_RX_VIDEO_HEIGHT : natural := 360;
   constant C_TX_VIDEO_WIDTH  : natural := 640;
   constant C_TX_VIDEO_HEIGHT : natural := 360;
   constant C_DATA_LENGTH     : natural := C_RX_VIDEO_WIDTH*C_RX_VIDEO_HEIGHT;
   constant C_EXPECT_LENGTH   : natural := C_TX_VIDEO_WIDTH*C_TX_VIDEO_HEIGHT;


   procedure wait_for_time_wrap(   -- Wait for next round time number - e.g. if now=2100ns, and round_time=1000ns, then next round time is 3000ns
      round_time    : time) is
      variable v_overshoot    : time    := now rem round_time;
   begin
      wait for (round_time - v_overshoot);
   end;

begin
   -----------------------------------------------------------------------------
   -- Instantiate test harness, containing DUT and Executors
   -----------------------------------------------------------------------------
   i_test_harness : entity work.th_scaler_complete
   generic map (
      g_data_width      => C_DATA_WIDTH,
      g_empty_width     => C_EMPTY_WIDTH,
      g_fifo_data_width => C_FIFO_DATA_WIDTH,
      g_fifo_data_depth => C_FIFO_DATA_DEPTH,
      --g_rx_video_width  => C_RX_VIDEO_WIDTH,
      --g_rx_video_height => C_RX_VIDEO_HEIGHT,
      g_tx_video_width  => C_TX_VIDEO_WIDTH,
      g_tx_video_height => C_TX_VIDEO_HEIGHT
   );
```

```
86
87
88    ------------------------------------------------
89    -- PROCESS: p_main
90    ------------------------------------------------
91    p_main: process
92       variable v_ctrl_pkt_array  : t_slv_array(0 to 1)(C_DATA_WIDTH-1 downto 0)                 := (others => (others
           ↪  => '0'));
93       variable v_data_array      : t_slv_array(0 to C_DATA_LENGTH)(C_DATA_WIDTH-1 downto 0)   := (others => (others =>
           ↪  '0'));
94       variable v_exp_data_array  : t_slv_array(0 to C_EXPECT_LENGTH-1)(C_DATA_WIDTH-1 downto 0) := (others => (others
           ↪  => '0'));
95       variable v_empty           : std_logic_vector(C_EMPTY_WIDTH-1 downto 0) := (others => '0');
96
97       variable v_num_test_loops  : natural := 0;
98
99       variable v_rx_video_width   : std_logic_vector(15 downto 0) := (others => '0');
100      variable v_rx_video_height  : std_logic_vector(15 downto 0) := (others => '0');
101
102      variable v_file_input_line    : line;
103      variable v_file_expect_line   : line;
104      variable v_file_data_input    : std_logic_vector(C_DATA_WIDTH-1 downto 0);
105      variable v_file_data_expect   : std_logic_vector(C_DATA_WIDTH-1 downto 0);
106
107      variable v_counter : integer := 0;
108   begin
109
110   -- Wait for UVVM to finish initialization
111   await_uvvm_initialization(VOID);
112
113   -- Print the configuration to the log
114   report_global_ctrl(VOID);
115   report_msg_id_panel(VOID);
116
117   -----------------------------------------------------------------------------
118   -- Enable log message
119   -----------------------------------------------------------------------------
120   enable_log_msg(ALL_MESSAGES);
121   enable_log_msg(ID_LOG_HDR);
122   enable_log_msg(ID_UVVM_SEND_CMD);
123
124   disable_log_msg(AVALON_ST_VVCT, 1, TX, ALL_MESSAGES);
125   disable_log_msg(AVALON_ST_VVCT, 1, RX, ALL_MESSAGES);
126
127   enable_log_msg(AVALON_ST_VVCT, 1, TX, ID_BFM);
128   enable_log_msg(AVALON_ST_VVCT, 1, TX, ID_PACKET_INITIATE);
129   enable_log_msg(AVALON_ST_VVCT, 1, TX, ID_PACKET_COMPLETE);
130
131   enable_log_msg(AVALON_ST_VVCT, 1, RX, ID_BFM);
132   enable_log_msg(AVALON_ST_VVCT, 1, RX, ID_PACKET_INITIATE);
133   enable_log_msg(AVALON_ST_VVCT, 1, RX, ID_PACKET_COMPLETE);
134
135   -----------------------------------------------------------------------------
136   -- Enable/disable Avalon-ST signals
137   -----------------------------------------------------------------------------
138   shared_avalon_st_vvc_config(TX, 1).bfm_config.use_channel   := false;
139   shared_avalon_st_vvc_config(TX, 1).bfm_config.use_error     := false;
140   shared_avalon_st_vvc_config(TX, 1).bfm_config.use_empty     := true;
141
142   -- Percent of cycles the receive module should assert ready_o signal
143   shared_avalon_st_vvc_config(RX, 1).bfm_config.ready_percentage := 100;
144
145   -- Set empty signal if some symbols are empty at the last transmission
146   v_empty := std_logic_vector(to_unsigned(0, v_empty'length));
147
148
149   log(ID_LOG_HDR, "Starting simulation of TB scaler", C_SCOPE);
150   log("Wait 10 clock period for reset to be turned off");
151   wait for (10 * C_CLK_PERIOD);
152
153
154   -----------------------------------------------------------------------------
155   -- Control packet
156   -----------------------------------------------------------------------------
157   log(ID_LOG_HDR, "Sending control packet", C_SCOPE);
158   -- Send control packet
159   v_ctrl_pkt_array(0) := std_logic_vector(to_unsigned(15, C_DATA_WIDTH));
160
161   -- Set rx resolution
162   v_rx_video_width  := std_logic_vector(to_unsigned(C_RX_VIDEO_WIDTH, v_rx_video_width'length));
163   v_rx_video_height := std_logic_vector(to_unsigned(C_RX_VIDEO_HEIGHT, v_rx_video_height'length));
164   v_ctrl_pkt_array(1)(3 downto 0)   := v_rx_video_width(15 downto 12);
165   v_ctrl_pkt_array(1)(13 downto 10) := v_rx_video_width(11 downto 8);
166   v_ctrl_pkt_array(1)(23 downto 20) := v_rx_video_width(7 downto 4);
167   v_ctrl_pkt_array(1)(33 downto 30) := v_rx_video_width(3 downto 0);
168   v_ctrl_pkt_array(1)(43 downto 40) := v_rx_video_height(15 downto 12);
169   v_ctrl_pkt_array(1)(53 downto 50) := v_rx_video_height(11 downto 8);
170   v_ctrl_pkt_array(1)(63 downto 60) := v_rx_video_height(7 downto 4);
171   v_ctrl_pkt_array(1)(73 downto 70) := v_rx_video_height(3 downto 0);
172
```

```
173        -- Start send and receive VVC
174        avalon_st_send(AVALON_ST_VVCT, 1, v_ctrl_pkt_array, v_empty, "Sending v_data_array");
175        --avalon_st_receive(AVALON_ST_VVCT, 1, "Receiving");
176        avalon_st_expect(AVALON_ST_VVCT, 1, v_ctrl_pkt_array, v_empty, "Checking data", ERROR);
177
178        -- Wait for completion
179        await_completion(AVALON_ST_VVCT, 1, RX, 10*C_DATA_LENGTH*C_CLK_PERIOD);
180        wait for (10 * C_CLK_PERIOD);
181
182        ----------------------------------------------------------------------------
183        -- Video data packet
184        ----------------------------------------------------------------------------
185        log(ID_LOG_HDR, "Sending video data packet", C_SCOPE);
186        -- Number of times to run the test loop
187        v_num_test_loops := 1;
188
189        --wait_for_time_wrap(10000 ns);
190
191        for i in 1 to v_num_test_loops loop
192           -- Create a random ready percentage for the recieve module
193           shared_avalon_st_vvc_config(RX, 1).bfm_config.ready_percentage := random(1,100);
194           --shared_avalon_st_vvc_config(RX, 1).bfm_config.ready_percentage := 50;
195
196           -- Write packet info, Data[3:0] = 0 for video_packet
197           v_data_array(0)      := std_logic_vector(to_unsigned(0, C_DATA_WIDTH));
198           v_exp_data_array(0)  := std_logic_vector(to_unsigned(0, C_DATA_WIDTH));
199
200           --------------------------------------------------
201           -- Read input file and fill data array
202           --------------------------------------------------
203           file_open(file_input, C_INPUT_FILE, read_mode);
204
205           while not endfile(file_input) loop
206              v_counter := v_counter + 1;
207
208              -- Read input data and store to data array
209              readline(file_input, v_file_input_line);
210              read(v_file_input_line, v_file_data_input);
211              v_data_array(v_counter) := v_file_data_input;
212           end loop;
213
214           file_close(file_input);
215
216           -- Reset v_counter
217           v_counter := 0;
218
219
220           --------------------------------------------------
221           -- Read expect file and fill expect data array
222           --------------------------------------------------
223           file_open(file_expect, C_EXPECT_FILE, read_mode);
224
225           while not endfile(file_expect) loop
226              -- Read expected output data and store to expect array
227              readline(file_expect, v_file_expect_line);
228              read(v_file_expect_line, v_file_data_expect);
229              v_exp_data_array(v_counter) := v_file_data_expect;
230              v_counter := v_counter + 1;
231           end loop;
232
233           file_close(file_expect);
234
235           -- Reset v_counter
236           v_counter := 0;
237
238           -- Margin
239           wait for 10*C_CLK_PERIOD;
240
241
242           --------------------------------------------------
243           -- Send/recieve using avalon st vvc
244           --------------------------------------------------
245
246           log(ID_LOG_HDR, "Test loop " & to_string(i) & " of " & to_string(v_num_test_loops) & " tests. Sending " &
              ↪    to_string(C_DATA_LENGTH) & " pixels. Using ready percentage: " &
              ↪    to_string(shared_avalon_st_vvc_config(RX, 1).bfm_config.ready_percentage), C_SCOPE);
247
248           -- Start send and receive VVC
249           avalon_st_send(AVALON_ST_VVCT, 1, v_data_array, v_empty, "Sending v_data_array");
250           --avalon_st_receive(AVALON_ST_VVCT, 1, "Receiving");
251           avalon_st_expect(AVALON_ST_VVCT, 1, v_exp_data_array, v_empty, "Checking data", ERROR);
252
253
254           -- Wait for completion
255           await_completion(AVALON_ST_VVCT, 1, RX, 100*C_DATA_LENGTH*C_CLK_PERIOD);
256        end loop;
257
258
259        ----------------------------------------------------------------------------
260        -- Ending the simulation
```

```
261       ----------------------------------------------------------------------------
262       wait for 1000 ns;             -- to allow some time for completion
263       report_alert_counters(FINAL); -- Report final counters and print conclusion for simulation (Success/Fail)
264       log(ID_LOG_HDR, "SIMULATION COMPLETED", C_SCOPE);
265
266       -- Finish the simulation
267       std.env.stop;
268       wait;  -- to stop completely
269
270       end process p_main;
271    end func;
```

B.6 Scaler Top Level Testharness With UVVM

```vhdl
-------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-01-21
-- Version: 0.1
-------------------------------------------------------
-- Description: Testharness for scaler full design
-------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;

library vip_avalon_st;


-- Test harness entity
entity th_scaler_complete is
   generic (
         g_data_width                : natural;
         g_empty_width               : natural;
         g_fifo_data_width           : natural;
         g_fifo_data_depth           : natural;
         --g_rx_video_width            : natural;
         --g_rx_video_height           : natural;
         g_tx_video_width            : natural;
         g_tx_video_height           : natural
      );
end entity;

-- Test harness architecture
architecture struct of th_scaler_complete is
   -- DSP interface and general control signals
   signal clk_i                : std_logic  := '0';
   signal sreset_i             : std_logic := '0';

   -- DUT scaler inputs
   signal startofpacket_i      : std_logic;
   signal endofpacket_i        : std_logic;
   signal data_i               : std_logic_vector(g_data_width-1 downto 0);
   signal empty_i              : std_logic_vector(g_empty_width-1 downto 0);
   signal valid_i              : std_logic;
   signal ready_i              : std_logic;
   -- DUT scaler outputs
   signal startofpacket_o      : std_logic := '0';
   signal endofpacket_o        : std_logic := '0';
   signal data_o               : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal empty_o              : std_logic_vector(g_empty_width-1 downto 0) := (others => '0');
   signal valid_o              : std_logic := '0';
   signal ready_o              : std_logic := '0';

   -- Sink
   signal sink_startofpacket_i      : std_logic;
   signal sink_endofpacket_i        : std_logic;
   signal sink_data_i               : std_logic_vector(g_data_width-1 downto 0);
   signal sink_empty_i              : std_logic_vector(g_empty_width-1 downto 0);
   signal sink_valid_i              : std_logic;
   signal sink_ready_o              : std_logic := '0';

   -- Source
   signal source_startofpacket_o      : std_logic := '0';
   signal source_endofpacket_o        : std_logic := '0';
   signal source_data_o               : std_logic_vector(g_data_width-1 downto 0) := (others => '0');
   signal source_empty_o              : std_logic_vector(g_empty_width-1 downto 0) := (others => '0');
   signal source_valid_o              : std_logic := '0';
   signal source_ready_i              : std_logic;

   constant C_CLK_PERIOD       : time := 10 ns; -- 100 MHz
begin
   -----------------------------------------------------------------------------
   -- Instantiate the concurrent procedure that initializes UVVM
   -----------------------------------------------------------------------------
   i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;

   -----------------------------------------------------------------------------
   -- Instantiate DUT
   -----------------------------------------------------------------------------
   i_scaler: entity work.scaler_wrapper
```

```
87        generic map (
88           g_data_width      => g_data_width,
89           g_empty_width     => g_empty_width,
90           g_fifo_data_width => g_fifo_data_width,
91           g_fifo_data_depth => g_fifo_data_depth,
92           g_tx_video_width  => g_tx_video_width,
93           g_tx_video_height => g_tx_video_height,
94           g_tx_video_scaling_method => 1
95        )
96        port map (
97           clk_i             => clk_i,
98           sreset_i          => sreset_i,
99
100          -- x -> scaler
101          data_i            => data_i,
102          ready_o           => ready_o,
103          valid_i           => valid_i,
104          empty_i           => empty_i,
105          endofpacket_i     => endofpacket_i,
106          startofpacket_i   => startofpacket_i,
107
108          -- scaler -> x
109          data_o            => data_o,
110          ready_i           => ready_i,
111          valid_o           => valid_o,
112          empty_o           => empty_o,
113          endofpacket_o     => endofpacket_o,
114          startofpacket_o   => startofpacket_o
115       );
116
117
118       ----------------------------------------------------------------------------
119       -- AVALON ST VVC
120       ----------------------------------------------------------------------------
121       i1_avalon_st_vvc: entity vip_avalon_st.avalon_st_vvc
122       generic map(
123          GC_DATA_WIDTH     => g_data_width,
124          GC_EMPTY_WIDTH    => g_empty_width,
125          GC_INSTANCE_IDX   => 1
126       )
127       port map(
128          clk   => clk_i,
129
130          -- Sink
131          avalon_st_sink_if.data_i               => sink_data_i,
132          avalon_st_sink_if.ready_o              => sink_ready_o,
133          avalon_st_sink_if.valid_i              => sink_valid_i,
134          avalon_st_sink_if.empty_i              => sink_empty_i,
135          avalon_st_sink_if.endofpacket_i        => sink_endofpacket_i,
136          avalon_st_sink_if.startofpacket_i      => sink_startofpacket_i,
137
138          -- Source
139          avalon_st_source_if.data_o             => source_data_o,
140          avalon_st_source_if.ready_i            => source_ready_i,
141          avalon_st_source_if.valid_o            => source_valid_o,
142          avalon_st_source_if.empty_o            => source_empty_o,
143          avalon_st_source_if.endofpacket_o      => source_endofpacket_o,
144          avalon_st_source_if.startofpacket_o    => source_startofpacket_o
145       );
146
147
148       ----------------------------------------------------------------------------
149       -- Connect: source -> scaler -> sink
150       ----------------------------------------------------------------------------
151       data_i           <= source_data_o;
152       source_ready_i   <= ready_o;
153       valid_i          <= source_valid_o;
154       empty_i          <= source_empty_o;
155       endofpacket_i    <= source_endofpacket_o;
156       startofpacket_i  <= source_startofpacket_o;
157
158       sink_data_i             <= data_o;
159       ready_i                 <= sink_ready_o;
160       sink_valid_i            <= valid_o;
161       sink_empty_i            <= empty_o;
162       sink_endofpacket_i      <= endofpacket_o;
163       sink_startofpacket_i    <= startofpacket_o;
164
165
166
167       ----------------------------------------------------------------------------
168       -- Reset process
169       ----------------------------------------------------------------------------
170       -- Toggle the reset after 5 clock periods
171       p_sreset: sreset_i <= '1', '0' after 5 *C_CLK_PERIOD;
172
173       ----------------------------------------------------------------------------
174       -- Clock process
175       ----------------------------------------------------------------------------
176       p_clk: process
```

```
177        begin
178           clk_i <= '0', '1' after C_CLK_PERIOD / 2;
179           wait for C_CLK_PERIOD;
180        end process;
181
182     end struct;
```

Appendix C

Avalon-ST Verification IP source code

Due to the enormous size of the full Avalon-ST Verification IP implementation in UVVM, only the BFM (Bus Functional Model) is being presented here. For the full source code, visit the open-source repository of the implementation at:
https://github.com/thoste/UVVM/tree/thoste

C.1 Avalon-ST BFM

```
--========================================================================================================================
-- Copyright (c) 2019 by Thomas Stenseth.  All rights reserved.
-- You should have received a copy of the license file containing the MIT License (see LICENSE.TXT).
--========================================================================================================================

------------------------------------------------------------------------------------------------------------------------
-- Description   :
------------------------------------------------------------------------------------------------------------------------

--========================================================================================================================
-- This VVC was generated with Bitvis VVC Generator
--========================================================================================================================

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

--========================================================================================================================
package avalon_st_bfm_pkg is

   ↪   --========================================================================================================================
   -- Types and constants for AVALON-ST BFM
   ↪   --========================================================================================================================
   constant C_SCOPE : string := "AVALON-ST BFM";

   -- Avalon Interface signals
   type t_avalon_st_source_if is
   record
      --channel_o          : std_logic_vector;
      data_o             : std_logic_vector;
      --error_o            : std_logic_vector;
```

```
35          ready_i           : std_logic;
36          valid_o           : std_logic;
37          empty_o           : std_logic_vector;
38          endofpacket_o     : std_logic;
39          startofpacket_o   : std_logic;
40       end record;
41
42       type t_avalon_st_sink_if is
43       record
44          --channel_i         : std_logic_vector;
45          data_i            : std_logic_vector;
46          --error_i           : std_logic_vector;
47          ready_o           : std_logic;
48          valid_i           : std_logic;
49          empty_i           : std_logic_vector;
50          endofpacket_i     : std_logic;
51          startofpacket_i   : std_logic;
52          -- Debug signal
53          --check_data        : std_logic_vector;
54       end record;
55
56       -- Configuration record to be assigned in the test harness.
57       type t_avalon_st_bfm_config is
58       record
59          max_wait_cycles          : integer;                -- The maximum number of clock cycles to wait before reporting
            ↪    a timeout alert.
60          max_wait_cycles_severity : t_alert_level;          -- The above timeout will have this severity
61          clock_period             : time;                   -- Period of the clock signal
62          clock_period_margin      : time;                   -- Input clock period accuracy margin to specified
            ↪    clock_period
63          clock_margin_severity    : t_alert_level;          -- The above margin will have this severity
64          setup_time               : time;                   -- Setup time for generated signals, set to clock_period/4
65          hold_time                : time;                   -- Hold time for generated signals, set to clock_period/4
66          ready_percentage         : natural range 0 to 100; -- Percent of cycles the receive module should assert ready_o
            ↪    signal
67          -- Enable/disable specific signals in Alavlon-ST
68          use_channel              : boolean;          -- Use channel signal
69          use_error                : boolean;          -- Use error signal
70          use_empty                : boolean;          -- Use empty signal
71          -- Common
72          id_for_bfm               : t_msg_id;         -- The message ID used as a general message ID in the Avalon-ST BFM
73          id_for_bfm_wait          : t_msg_id;         -- The message ID used for logging waits in the Avalon-ST BFM
74          id_for_bfm_poll          : t_msg_id;         -- The message ID used for logging polling in the Avalon-ST BFM
75       end record;
76
77       constant C_AVALON_ST_BFM_CONFIG_DEFAULT : t_avalon_st_bfm_config := (
78          max_wait_cycles            => 10,
79          max_wait_cycles_severity   => failure,
80          clock_period               => 10 ns,
81          clock_period_margin        => 0 ns,
82          clock_margin_severity      => TB_ERROR,
83          setup_time                 => 2.5 ns,
84          hold_time                  => 2.5 ns,
85          ready_percentage           => 100,
86          use_channel                => false,
87          use_error                  => false,
88          use_empty                  => false,
89          id_for_bfm                 => ID_BFM,
90          id_for_bfm_wait            => ID_BFM_WAIT,
91          id_for_bfm_poll            => ID_BFM_POLL
92       );
93
94
95
      ↪    --========================================================================================================================
96       -- BFM procedures
97
      ↪    --========================================================================================================================
98
99       ------------------------------------------
100      -- init_avalon_st_source_if_signals
101      ------------------------------------------
102      function init_avalon_st_source_if_signals (
103         --channel_width  : natural;
104         data_width     : natural;
105         --error_width    : natural;
106         empty_width    : natural
107      ) return t_avalon_st_source_if;
108
109      ------------------------------------------
110      -- init_avalon_st_sink_if_signals
111      ------------------------------------------
112      function init_avalon_st_sink_if_signals (
113         --channel_width  : natural;
114         data_width     : natural;
115         --error_width    : natural;
116         empty_width    : natural
117      ) return t_avalon_st_sink_if;
118
119      --------------------------------------------------------
```

```
120    --
121    -- Avalon-ST send
122    --
123    ----------------------------------------------------------
124    procedure avalon_st_send (
125       --constant channel_num          : in     std_logic_vector;
126       constant data_array           : in     t_slv_array;
127       constant data_width           : in     natural;
128       --constant error_bit_mask       : in     t_slv_array;
129       constant empty                : in     std_logic_vector;
130       constant empty_width          : in     natural;
131       constant msg                  : in     string;
132       signal   clk                  : in     std_logic;
133       signal   avalon_st_source_if  : inout  t_avalon_st_source_if;
134       constant scope                : in     string                    := C_SCOPE;
135       constant msg_id_panel         : in     t_msg_id_panel            := shared_msg_id_panel;
136       constant config               : in     t_avalon_st_bfm_config   := C_AVALON_ST_BFM_CONFIG_DEFAULT
137    );
138
139    --------------------------------------------------------------------------------
140    -- Overloaded version without records
141    --------------------------------------------------------------------------------
142    procedure avalon_st_send (
143       --constant channel_num       : in     std_logic_vector;
144       constant data_array        : in     t_slv_array;
145       constant data_width        : in     natural;
146       --constant error_bit_mask    : in     t_slv_array;
147       constant empty             : in     std_logic_vector;
148       constant empty_width       : in     natural;
149       constant msg               : in     string;
150       signal   clk               : in     std_logic;
151       signal   data_o            : inout  std_logic_vector;
152       signal   ready_i           : inout  std_logic;
153       signal   valid_o           : inout  std_logic;
154       signal   empty_o           : inout  std_logic_vector;
155       signal   endofpacket_o     : inout  std_logic;
156       signal   startofpacket_o   : inout  std_logic;
157       constant scope             : in     string                    := C_SCOPE;
158       constant msg_id_panel      : in     t_msg_id_panel            := shared_msg_id_panel;
159       constant config            : in     t_avalon_st_bfm_config   := C_AVALON_ST_BFM_CONFIG_DEFAULT
160    );
161
162    ----------------------------------------------------------
163    --
164    -- Avalon-ST receive
165    --
166    ----------------------------------------------------------
167    procedure avalon_st_receive (
168       --variable channel_num          : inout     std_logic_vector;
169       variable data_array           : inout  t_slv_array;
170       constant data_width           : in     natural;
171       variable data_length          : inout  natural;
172       --variable error_bit_mask       : inout     t_slv_array;
173       variable empty                : inout  std_logic_vector;
174       constant empty_width          : in     natural;
175       constant msg                  : in     string;
176       signal   clk                  : in     std_logic;
177       signal   avalon_st_sink_if    : inout  t_avalon_st_sink_if;
178       constant scope                : in     string                    := C_SCOPE;
179       constant msg_id_panel         : in     t_msg_id_panel            := shared_msg_id_panel;
180       constant config               : in     t_avalon_st_bfm_config   := C_AVALON_ST_BFM_CONFIG_DEFAULT
181    );
182
183    --------------------------------------------------------------------------------
184    -- Overloaded version without records
185    --------------------------------------------------------------------------------
186    procedure avalon_st_receive (
187       --variable channel_num          : inout     std_logic_vector;
188       variable data_array           : inout  t_slv_array;
189       constant data_width           : in     natural;
190       variable data_length          : inout  natural;
191       --variable error_bit_mask       : inout     t_slv_array;
192       variable empty                : inout  std_logic_vector;
193       constant empty_width          : in     natural;
194       constant msg                  : in     string;
195       signal   clk                  : in     std_logic;
196       signal   data_i               : inout  std_logic_vector;
197       signal   ready_o              : inout  std_logic;
198       signal   valid_i              : inout  std_logic;
199       signal   empty_i              : inout  std_logic_vector;
200       signal   endofpacket_i        : inout  std_logic;
201       signal   startofpacket_i      : inout  std_logic;
202       constant scope                : in     string                    := C_SCOPE;
203       constant msg_id_panel         : in     t_msg_id_panel            := shared_msg_id_panel;
204       constant config               : in     t_avalon_st_bfm_config   := C_AVALON_ST_BFM_CONFIG_DEFAULT
205    );
206
207    ----------------------------------------------------------
208    --
209    -- Avalon-ST expect
```

```
210      --
211      ----------------------------------------------------------
212      procedure avalon_st_expect (
213         constant exp_data_array    : in     t_slv_array;
214         constant exp_data_width    : in     natural;
215         constant exp_empty         : in     std_logic_vector;
216         constant exp_empty_width   : in     natural;
217         constant msg               : in     string;
218         signal   clk               : in     std_logic;
219         signal   avalon_st_sink_if : inout  t_avalon_st_sink_if;
220         constant alert_level       : in     t_alert_level          := error;
221         constant scope             : in     string                 := C_SCOPE;
222         constant msg_id_panel      : in     t_msg_id_panel         := shared_msg_id_panel;
223         constant config            : in     t_avalon_st_bfm_config := C_AVALON_ST_BFM_CONFIG_DEFAULT
224      );
225
226      ------------------------------------------------------------------------------------
227      -- Overloaded version without records
228      ------------------------------------------------------------------------------------
229      procedure avalon_st_expect (
230         constant exp_data_array    : in     t_slv_array;
231         constant exp_data_width    : in     natural;
232         constant exp_empty         : in     std_logic_vector;
233         constant exp_empty_width   : in     natural;
234         constant msg               : in     string;
235         signal   clk               : in     std_logic;
236         signal   data_i            : inout std_logic_vector;
237         signal   ready_o           : inout std_logic;
238         signal   valid_i           : inout std_logic;
239         signal   empty_i           : inout std_logic_vector;
240         signal   endofpacket_i     : inout std_logic;
241         signal   startofpacket_i   : inout std_logic;
242         constant alert_level       : in     t_alert_level          := error;
243         constant scope             : in     string                 := C_SCOPE;
244         constant msg_id_panel      : in     t_msg_id_panel         := shared_msg_id_panel;
245         constant config            : in     t_avalon_st_bfm_config := C_AVALON_ST_BFM_CONFIG_DEFAULT
246      );
247
248   end package avalon_st_bfm_pkg;
249
250
251   --==========================================================================================================================
252   --==========================================================================================================================
253
254   package body avalon_st_bfm_pkg is
255
256      ------------------------------------------------------------------------------------
257      -- initialize Avalon-ST to DUT signals
258      ------------------------------------------------------------------------------------
259      function init_avalon_st_source_if_signals (
260            --channel_width  : natural;
261            data_width     : natural;
262            --error_width    : natural;
263            empty_width    : natural
264         ) return t_avalon_st_source_if is
265         variable result : t_avalon_st_source_if(
266            --channel_o(channel_width - 1 downto 0),
267            data_o(data_width - 1 downto 0),
268            --error_o(error_width -1 downto 0),
269            empty_o(empty_width -1 downto 0)
270         );
271      begin
272         --result.channel_o        := (result.channel_o'range    => '0');
273         result.data_o           := (result.data_o'range       => '0');
274         --result.error_o          := (result.error_o'range      => '0');
275         result.ready_i          := 'Z';
276         result.valid_o          := '0';
277         result.empty_o          := (result.empty_o'range      => '0');
278         result.endofpacket_o    := '0';
279         result.startofpacket_o  := '0';
280         return result;
281      end function;
282
283      ------------------------------------------------------------------------------------
284      -- initialize DUT to Avalon-ST signals
285      ------------------------------------------------------------------------------------
286      function init_avalon_st_sink_if_signals (
287            --channel_width  : natural;
288            data_width     : natural;
289            --error_width    : natural;
290            empty_width    : natural
291         ) return t_avalon_st_sink_if is
292         variable result : t_avalon_st_sink_if(
293            --channel_i(channel_width - 1 downto 0),
294            data_i(data_width - 1 downto 0),
295            --error_i(error_width -1 downto 0),
296            empty_i(empty_width -1 downto 0)
297            --check_data(data_width -1 downto 0)
298         );
299      begin
```

```
300         --result.channel_i        := (result.channel_i'range    => 'Z');
301         result.data_i           := (result.data_i'range       => 'Z');
302         --result.error_i          := (result.error_i'range      => 'Z');
303         result.ready_o          := '0';
304         result.valid_i          := 'Z';
305         result.empty_i          := (result.empty_i'range      => 'Z');
306         result.endofpacket_i    := 'Z';
307         result.startofpacket_i  := 'Z';
308         return result;
309      end function;
310
311      --------------------------------------------------------
312      --
313      -- Avalon-ST send
314      --
315      --------------------------------------------------------
316      procedure avalon_st_send (
317         --constant channel_num          : in     std_logic_vector;
318         constant data_array           : in     t_slv_array;
319         constant data_width           : in     natural;
320         --constant error_bit_mask       : in     t_slv_array;
321         constant empty                : in     std_logic_vector;
322         constant empty_width          : in     natural;
323         constant msg                  : in     string;
324         signal   clk                  : in     std_logic;
325         signal   avalon_st_source_if  : inout  t_avalon_st_source_if;
326         constant scope                : in     string                  := C_SCOPE;
327         constant msg_id_panel         : in     t_msg_id_panel          := shared_msg_id_panel;
328         constant config               : in     t_avalon_st_bfm_config  := C_AVALON_ST_BFM_CONFIG_DEFAULT
329      ) is
330         constant proc_name : string := "avalon_st_send";
331         --onstant proc_call : string := proc_name & "(" & to_string(data_array, HEX, AS_IS, INCL_RADIX) & ")";
332         constant proc_call : string := proc_name & "(" & to_string(data_width) & " bits)";
333
334         variable v_tot_symbols : natural;
335         variable v_top_sym : natural;
336         variable v_bot_sym : natural;
337         variable v_empty : natural;
338      begin
339         check_value(data_array'ascending, TB_ERROR, "Sanity check: Check that data_array is ascending (defined with
            ↪    'to'), for byte order clarity", scope, ID_NEVER, msg_id_panel, proc_call);
340
341         -- setup_time and hold_time checking
342         check_value(config.setup_time < config.clock_period/2, TB_FAILURE, "Sanity check: Check that setup_time do not
            ↪    exceed clock_period/2.", scope, ID_NEVER, msg_id_panel, proc_call);
343         check_value(config.hold_time < config.clock_period/2, TB_FAILURE, "Sanity check: Check that hold_time do not
            ↪    exceed clock_period/2.", scope, ID_NEVER, msg_id_panel, proc_call);
344         check_value(config.setup_time > 0 ns, TB_FAILURE, "Sanity check: Check that setup_time is more than 0 ns.",
            ↪    scope, ID_NEVER, msg_id_panel, proc_call);
345         check_value(config.hold_time > 0 ns, TB_FAILURE, "Sanity check: Check that hold_time is more than 0 ns.", scope,
            ↪    ID_NEVER, msg_id_panel, proc_call);
346
347         -- check if enough room for setup_time in low period
348         if (clk = '0') and (config.setup_time > (config.clock_period/2 - clk'last_event))then
349            await_value(clk, '1', 0 ns, config.clock_period/2, TB_FAILURE, proc_name & ": timeout waiting for clk low
               ↪    period for setup_time.");
350         end if;
351         -- Wait setup_time specified in config record
352         wait_until_given_time_before_rising_edge(clk, config.setup_time, config.clock_period);
353
354         log(ID_PACKET_INITIATE, proc_call & " => " & add_msg_delimiter(msg), scope, msg_id_panel);
355         wait until rising_edge(clk);
356
357         -- Loop through data_array
358         for byte in 0 to data_array'high loop
359
360            -- Wait for ready signal
361            while (avalon_st_source_if.ready_i = '0') loop
362               -- Wait for next clock cycle, then check for ready
363               wait until rising_edge(clk);
364            end loop;
365
366            -- Check for start of data_array
367            if (byte = 0) then
368               -- Beginning of packet transmission, send startofpacket
369               log(ID_PACKET_DATA, proc_call & " => Sending startofpacket", scope, msg_id_panel);
370               avalon_st_source_if.startofpacket_o <= '1', '0' after config.clock_period;
371            end if;
372
373            ---- Set channel to recieve data
374            --if (config.use_channel) then
375            --   avalon_st_source_if.channel_o <= channel_num;
376            --end if;
377
378            ---- Send error bit mask
379            --if (config.use_error) then
380            --   avalon_st_source_if.error_o <= error_bit_mask(byte);
381            --end if;
382
383            -- Send symbols to data_o
```

```
384          log(ID_PACKET_DATA, proc_call & " => TX: " & to_string(data_array(byte)(data_width-1 downto 0), HEX, AS_IS,
               ↪    INCL_RADIX) & ", array entry# " & to_string(byte) & ". " & add_msg_delimiter(msg), scope,
               ↪    msg_id_panel);
385          avalon_st_source_if.data_o <= data_array(byte)(data_width-1 downto 0);
386          avalon_st_source_if.valid_o <= '1';
387
388          -- Check for end of data_array
389          if byte = data_array'high then
390             -- Packet done
391             if (config.use_empty) then
392                -- Send empty signal together with last data
393                log(ID_PACKET_DATA, proc_call & " => Number of symbols that are empty: " & to_string(empty, DEC, AS_IS,
                     ↪    INCL_RADIX), scope, msg_id_panel);
394                avalon_st_source_if.empty_o <= empty(empty_width-1 downto 0);
395             end if;
396             -- Send endofpacket
397             log(ID_PACKET_DATA, proc_call & " => Sending endofpacket", scope, msg_id_panel);
398             --avalon_st_source_if.endofpacket_o <= '1', '0' after config.clock_period;
399             --avalon_st_source_if.valid_o <= '1', '0' after config.clock_period;
400             avalon_st_source_if.endofpacket_o <= '1';
401             avalon_st_source_if.valid_o <= '1';
402          end if;
403          wait until rising_edge(clk);
404       end loop;
405
406       -- Wait until module is done recieving
407       while (avalon_st_source_if.ready_i = '0') loop
408          -- Wait for next clock cycle, then check for ready
409          wait until rising_edge(clk);
410       end loop;
411
412       log(ID_PACKET_COMPLETE, proc_call & " => Sent " & to_string(data_array'high + 1) & " data entries", scope,
            ↪    msg_id_panel);
413
414       -- Done, set avalon_st_source_if back to default
415       avalon_st_source_if <= init_avalon_st_source_if_signals(
416          data_width  => avalon_st_source_if.data_o'length,
417          empty_width => avalon_st_source_if.empty_o'length
418       );
419    end procedure avalon_st_send;
420
421
422    --------------------------------------------------------------------------------
423    -- Overloaded version without records
424    --------------------------------------------------------------------------------
425    procedure avalon_st_send (
426       --constant channel_num       : in     std_logic_vector;
427       constant data_array        : in     t_slv_array;
428       constant data_width        : in     natural;
429       --constant error_bit_mask    : in     t_slv_array;
430       constant empty             : in     std_logic_vector;
431       constant empty_width       : in     natural;
432       constant msg               : in     string;
433       signal   clk               : in     std_logic;
434       signal   data_o            : inout  std_logic_vector;
435       signal   ready_i           : inout  std_logic;
436       signal   valid_o           : inout  std_logic;
437       signal   empty_o           : inout  std_logic_vector;
438       signal   endofpacket_o     : inout  std_logic;
439       signal   startofpacket_o   : inout  std_logic;
440       constant scope             : in     string                  := C_SCOPE;
441       constant msg_id_panel      : in     t_msg_id_panel          := shared_msg_id_panel;
442       constant config            : in     t_avalon_st_bfm_config  := C_AVALON_ST_BFM_CONFIG_DEFAULT
443    ) is
444    begin
445       -- Simply call the record version
446       avalon_st_send(
447          data_array         => data_array,
448          data_width         => data_width,
449          empty              => empty,
450          empty_width        => empty_width,
451          msg                => msg,
452          clk                => clk,
453          avalon_st_source_if.data_o            => data_o,
454          avalon_st_source_if.ready_i           => ready_i,
455          avalon_st_source_if.valid_o           => valid_o,
456          avalon_st_source_if.empty_o           => empty_o,
457          avalon_st_source_if.endofpacket_o     => endofpacket_o,
458          avalon_st_source_if.startofpacket_o   => startofpacket_o,
459          scope          => scope,
460          msg_id_panel   => msg_id_panel,
461          config         => config
462       );
463    end procedure avalon_st_send;
464
465
466    ---------------------------------------------------------
467    --
468    -- Avalon-ST receive
469    --
```

```
470     --------------------------------------------------------
471     procedure avalon_st_receive (
472        --variable channel_num          : inout     std_logic_vector;
473        variable data_array           : inout     t_slv_array;
474        constant data_width           : in      natural;
475        variable data_length          : inout   natural;
476        --variable error_bit_mask       : inout     t_slv_array;
477        variable empty                : inout   std_logic_vector;
478        constant empty_width          : in      natural;
479        constant msg                  : in      string;
480        signal   clk                  : in      std_logic;
481        signal   avalon_st_sink_if    : inout   t_avalon_st_sink_if;
482        constant scope                : in      string                  := C_SCOPE;
483        constant msg_id_panel         : in      t_msg_id_panel          := shared_msg_id_panel;
484        constant config               : in      t_avalon_st_bfm_config  := C_AVALON_ST_BFM_CONFIG_DEFAULT
485     ) is
486        constant proc_name : string := "avalon_st_receive";
487        constant proc_call : string := proc_name & "(" & to_string(data_width) & " bits)";
488
489        variable v_ready_low_done  : boolean   := false;
490        variable v_received_sop    : boolean   := false;
491        variable v_done            : boolean   := false;
492        variable v_byte            : natural   := 0;
493
494        variable v_data_i          : std_logic_vector(19 downto 0) := (others => '0');
495        variable v_valid_i       : std_logic := '0';
496     begin
497        check_value(data_array'ascending, TB_ERROR, "Sanity check: Check that data_array is ascending (defined with
          ↪    'to'), for byte order clarity", scope, ID_NEVER, msg_id_panel, proc_call);
498
499        -- setup_time and hold_time checking
500        check_value(config.setup_time < config.clock_period/2, TB_FAILURE, "Sanity check: Check that setup_time do not
          ↪    exceed clock_period/2.", scope, ID_NEVER, msg_id_panel, proc_call);
501        check_value(config.hold_time < config.clock_period/2, TB_FAILURE, "Sanity check: Check that hold_time do not
          ↪    exceed clock_period/2.", scope, ID_NEVER, msg_id_panel, proc_call);
502        check_value(config.setup_time > 0 ns, TB_FAILURE, "Sanity check: Check that setup_time is more than 0 ns.",
          ↪    scope, ID_NEVER, msg_id_panel, proc_call);
503        check_value(config.hold_time > 0 ns, TB_FAILURE, "Sanity check: Check that hold_time is more than 0 ns.", scope,
          ↪    ID_NEVER, msg_id_panel, proc_call);
504
505        -- check if enough room for setup_time in low period
506        if (clk = '0') and (config.setup_time > (config.clock_period/2 - clk'last_event))then
507           await_value(clk, '1', 0 ns, config.clock_period/2, TB_FAILURE, proc_name & ": timeout waiting for clk low
             ↪    period for setup_time.");
508        end if;
509        -- Wait setup_time specified in config record
510        wait_until_given_time_before_rising_edge(clk, config.setup_time, config.clock_period);
511        log(ID_PACKET_INITIATE, proc_call & " => " & add_msg_delimiter(msg), scope, msg_id_panel);
512
513        ------------------------------------------------------------------------------------------------------------
514        -- Sample byte by byte until receive is done, (until endofpacket is received)
515        ------------------------------------------------------------------------------------------------------------
516        while not v_done loop
517           -- Hold module before asserting ready
518           if not v_ready_low_done then
519              wait_until_given_time_after_rising_edge(clk, config.hold_time);
520              v_ready_low_done := true;
521           end if;
522
523           -- Assert signals on rising edge
524           wait until rising_edge(clk);
525
526           if (random(1,100) <= config.ready_percentage) then
527              -- Signal that the module is ready to recieve
528              avalon_st_sink_if.ready_o <= '1';
529           else
530              avalon_st_sink_if.ready_o <= '0';
531           end if;
532
533           -- Wait for start of packet on a valid signal
534           if (avalon_st_sink_if.startofpacket_i = '1') and (avalon_st_sink_if.valid_i = '1') then
535              v_received_sop := true;
536              log(ID_PACKET_DATA, proc_call & " => Received startofpacket", scope, msg_id_panel);
537           end if;
538
539           -- Sample data packet on each valid signal until endofpacket is recieved
540           if v_received_sop and (avalon_st_sink_if.valid_i = '1') and avalon_st_sink_if.ready_o = '1' then
541              -- TODO: add last ready = 1 check, else assert error
542              data_array(v_byte)(data_width-1 downto 0) := avalon_st_sink_if.data_i;
543              log(ID_PACKET_DATA, proc_call & " => RX: " & to_string(data_array(v_byte)(data_width-1 downto 0), HEX,
                ↪    AS_IS, INCL_RADIX) & ", data_array entry# " & to_string(v_byte) & ". " & add_msg_delimiter(msg),
                ↪    scope, msg_id_panel);
544
545              if(avalon_st_sink_if.endofpacket_i = '1') then
546                 log(ID_PACKET_DATA, proc_call & " => Received endofpacket", scope, msg_id_panel);
547                 -- Empty signal received together with last data
548                 empty(empty_width-1 downto 0) := avalon_st_sink_if.empty_i;
549                 log(ID_PACKET_DATA, proc_call & " => RX empty signal: " & to_string(empty(empty_width-1 downto 0), HEX,
                   ↪    AS_IS, INCL_RADIX), scope, msg_id_panel);
550
```

```
551                 -- DANGEROUS!!!!
552                 -- Done receiving data
553                 v_done := true;
554                 -- Signal that module is not ready to receive any more data
555                 avalon_st_sink_if.ready_o <= '0';
556                 -- TODO:
557                 -- May lose data if new data arrives on next clock cycle
558                 -- This happens since endofpacket_i and data_i is sampled on current clk, but ready_o is set low on next
                      ↪    clk cycle
559                 -- Avalon-ST with ready latency of 1 (Avalon-ST video) implies that a module should be able to recieve
                      ↪    data one clk cycle after setting ready low
560              else
561                 -- Increase counter for data_array for next data to be received
562                 v_byte := v_byte + 1;
563              end if;
564           end if;
565
566        end loop;
567
568        data_length := v_byte;
569        log(ID_PACKET_COMPLETE, proc_call & " => Recieved " & to_string(data_length + 1) & " data entires", scope,
             ↪    msg_id_panel);
570
571        -- Done, set avalon_st_sink_if back to default
572        avalon_st_sink_if <= init_avalon_st_sink_if_signals(
573           data_width  => avalon_st_sink_if.data_i'length,
574           empty_width => avalon_st_sink_if.empty_i'length
575        );
576     end procedure avalon_st_receive;
577
578
579     ---------------------------------------------------------------------------------
580     -- Overloaded version without records
581     ---------------------------------------------------------------------------------
582     procedure avalon_st_receive (
583        --variable channel_num          : inout     std_logic_vector;
584        variable data_array           : inout     t_slv_array;
585        constant data_width           : in      natural;
586        variable data_length          : inout   natural;
587        --variable error_bit_mask       : inout     t_slv_array;
588        variable empty                : inout   std_logic_vector;
589        constant empty_width          : in      natural;
590        constant msg                  : in      string;
591        signal   clk                  : in      std_logic;
592        signal   data_i               : inout   std_logic_vector;
593        signal   ready_o              : inout   std_logic;
594        signal   valid_i              : inout   std_logic;
595        signal   empty_i              : inout   std_logic_vector;
596        signal   endofpacket_i        : inout   std_logic;
597        signal   startofpacket_i      : inout   std_logic;
598        constant scope                : in      string                   := C_SCOPE;
599        constant msg_id_panel         : in      t_msg_id_panel           := shared_msg_id_panel;
600        constant config               : in      t_avalon_st_bfm_config   := C_AVALON_ST_BFM_CONFIG_DEFAULT
601     ) is
602     begin
603        -- Simply call the record version
604        avalon_st_receive(
605           data_array                          => data_array,
606           data_length                         => data_length,
607           data_width                          => data_width,
608           empty                               => empty,
609           empty_width                         => empty_width,
610           msg                                 => msg,
611           clk                                 => clk,
612           avalon_st_sink_if.data_i            => data_i,
613           avalon_st_sink_if.ready_o           => ready_o,
614           avalon_st_sink_if.valid_i           => valid_i,
615           avalon_st_sink_if.empty_i           => empty_i,
616           avalon_st_sink_if.endofpacket_i     => endofpacket_i,
617           avalon_st_sink_if.startofpacket_i   => startofpacket_i,
618           scope                               => scope,
619           msg_id_panel                        => msg_id_panel,
620           config                              => config
621        );
622     end procedure avalon_st_receive;
623
624
625     --------------------------------------------------------
626     --
627     -- Avalon-ST expect
628     --
629     --------------------------------------------------------
630     procedure avalon_st_expect (
631        constant exp_data_array    : in      t_slv_array;
632        constant exp_data_width    : in      natural;
633        constant exp_empty         : in      std_logic_vector;
634        constant exp_empty_width   : in      natural;
635        constant msg               : in      string;
636        signal   clk               : in      std_logic;
637        signal   avalon_st_sink_if : inout   t_avalon_st_sink_if;
```

```
638         constant alert_level        : in     t_alert_level           := error;
639         constant scope              : in     string                  := C_SCOPE;
640         constant msg_id_panel       : in     t_msg_id_panel          := shared_msg_id_panel;
641         constant config             : in     t_avalon_st_bfm_config := C_AVALON_ST_BFM_CONFIG_DEFAULT
642      ) is
643         constant proc_name : string := "avalon_st_expect";
644         constant proc_call : string := proc_name & "(" & to_string(exp_data_width) & " data bits, " &
              ↪    to_string(exp_empty_width) & " empty bits)";
645
646         variable v_config           : t_avalon_st_bfm_config := config;
647         variable v_rx_data_array    : t_slv_array(exp_data_array'range)(exp_data_array(0)'range);  -- received data
648         variable v_rx_data_length   : natural;
649         variable v_rx_data_width    : natural;
650         variable v_rx_empty_slv     : std_logic_vector(exp_empty'range);
651         variable v_rx_empty_width   : natural;
652         variable v_data_error_cnt   : natural := 0;
653         variable v_empty_error_cnt  : natural := 0;
654         variable v_first_errored_byte : natural;
655      begin
656         v_rx_data_width   := exp_data_width;
657         v_rx_empty_width  := exp_empty_width;
658
659         -- Receive and store data
660         avalon_st_receive(
661            data_array        => v_rx_data_array,
662            data_length       => v_rx_data_length,
663            data_width        => v_rx_data_width,
664            empty             => v_rx_empty_slv,
665            empty_width       => v_rx_empty_width,
666            msg               => msg,
667            clk               => clk,
668            data_i            => avalon_st_sink_if.data_i,
669            ready_o           => avalon_st_sink_if.ready_o,
670            valid_i           => avalon_st_sink_if.valid_i,
671            empty_i           => avalon_st_sink_if.empty_i,
672            endofpacket_i     => avalon_st_sink_if.endofpacket_i,
673            startofpacket_i   => avalon_st_sink_if.startofpacket_i,
674            scope             => scope,
675            msg_id_panel      => msg_id_panel,
676            config            => v_config
677         );
678
679         -- Check if each received bit matches the expected
680         -- Find and report the first errored byte
681         for byte in v_rx_data_array'high downto 0 loop
682            for i in v_rx_data_width-1 downto 0 loop
683               if (exp_data_array(byte)(i) = '-') or  -- Expected set to don't care, or
684                  (v_rx_data_array(byte)(i) = exp_data_array(byte)(i)) then  -- received value matches expected
685                  -- Check is OK
686               else
687                  -- Received byte does not match the expected byte
688                  --log(ID_PACKET_DATA, proc_call & "=> DATA NOT OK, checked " &
                    ↪    to_string(v_rx_data_array(byte)(v_rx_data_width-1 downto 0), HEX, AS_IS, INCL_RADIX) & " = " &
                    ↪    to_string(exp_data_array(byte)(v_rx_data_width-1 downto 0), HEX, AS_IS, INCL_RADIX) & msg,
                    ↪    scope, msg_id_panel);
689                  v_data_error_cnt     := v_data_error_cnt + 1;
690                  v_first_errored_byte := byte;
691               end if;
692            end loop;
693         end loop;
694
695         for j in v_rx_empty_width-1 downto 0 loop
696            if(exp_empty(j) = '-') or
697               (v_rx_empty_slv(j) = exp_empty(j)) then
698                  -- Check is OK
699            else
700               -- Received empty does not match expected empty
701               --log(ID_PACKET_DATA, proc_call & "=> EMPTY NOT OK, checked " & to_string(v_rx_empty_slv(v_rx_empty_width-1
                 ↪    downto 0), HEX, AS_IS, INCL_RADIX) & " = " & to_string(exp_empty(v_rx_empty_width-1 downto 0), HEX,
                 ↪    AS_IS, INCL_RADIX) & msg, scope, msg_id_panel);
702               v_empty_error_cnt     := v_empty_error_cnt + 1;
703            end if;
704         end loop;
705
706         -- No more than one alert per packet
707       if v_data_error_cnt /= 0 then
708         alert(alert_level, proc_call & "=> Failed in " & to_string(v_data_error_cnt) & " data bits. First mismatch in
              ↪    byte# " & to_string(v_first_errored_byte) & ". Was " &
              ↪    to_string(v_rx_data_array(v_first_errored_byte)(v_rx_data_width-1 downto 0), HEX, AS_IS, INCL_RADIX) & ".
              ↪    Expected " & to_string(exp_data_array(v_first_errored_byte)(v_rx_data_width-1 downto 0), HEX, AS_IS,
              ↪    INCL_RADIX) & "." & LF & add_msg_delimiter(msg), scope);
709       elsif v_empty_error_cnt /= 0 then
710         alert(alert_level, proc_call & "=> Failed in " & to_string(v_empty_error_cnt) & " empty bits. Was " &
              ↪    to_string(v_rx_empty_slv(v_rx_empty_width-1 downto 0), HEX, AS_IS, INCL_RADIX) & ". Expected " &
              ↪    to_string(exp_empty(v_rx_empty_width-1 downto 0), HEX, AS_IS, INCL_RADIX) & "." & LF &
              ↪    add_msg_delimiter(msg), scope);
711       else
712         log(config.id_for_bfm, proc_call & "=> OK, received " & to_string(v_rx_data_array'length) & " data entries. " &
              ↪    add_msg_delimiter(msg), scope, msg_id_panel);
713       end if;
```

```
714
715      end procedure avalon_st_expect;
716
717
718      --------------------------------------------------------------------------------
719      -- Overloaded version without records
720      --------------------------------------------------------------------------------
721      procedure avalon_st_expect (
722         constant exp_data_array    : in     t_slv_array;
723         constant exp_data_width    : in     natural;
724         constant exp_empty         : in     std_logic_vector;
725         constant exp_empty_width   : in     natural;
726         constant msg               : in     string;
727         signal   clk               : in     std_logic;
728         signal   data_i            : inout  std_logic_vector;
729         signal   ready_o           : inout  std_logic;
730         signal   valid_i           : inout  std_logic;
731         signal   empty_i           : inout  std_logic_vector;
732         signal   endofpacket_i     : inout  std_logic;
733         signal   startofpacket_i   : inout  std_logic;
734         constant alert_level       : in     t_alert_level          := error;
735         constant scope             : in     string                 := C_SCOPE;
736         constant msg_id_panel      : in     t_msg_id_panel         := shared_msg_id_panel;
737         constant config            : in     t_avalon_st_bfm_config := C_AVALON_ST_BFM_CONFIG_DEFAULT
738      ) is
739      begin
740         -- Simply call the record version
741         avalon_st_expect(
742            exp_data_array                   => exp_data_array,
743            exp_data_width                   => exp_data_width,
744            exp_empty                        => exp_empty,
745            exp_empty_width                  => exp_empty_width,
746            msg                              => msg,
747            clk                              => clk,
748            avalon_st_sink_if.data_i         => data_i,
749            avalon_st_sink_if.ready_o        => ready_o,
750            avalon_st_sink_if.valid_i        => valid_i,
751            avalon_st_sink_if.empty_i        => empty_i,
752            avalon_st_sink_if.endofpacket_i  => endofpacket_i,
753            avalon_st_sink_if.startofpacket_i => startofpacket_i,
754            scope                            => scope,
755            msg_id_panel                     => msg_id_panel,
756            config                           => config
757         );
758      end procedure avalon_st_expect;
759
760   end package body avalon_st_bfm_pkg;
```

C.2 Avalon-ST VVC Testbench

```vhdl
------------------------------------------------------------------------------------------
-- Project: FPGA video scaler
-- Author: Thomas Stenseth
-- Date: 2019-03-11
-- Version: 0.1
------------------------------------------------------------------------------------------
-- Description:
------------------------------------------------------------------------------------------
-- v0.1:
------------------------------------------------------------------------------------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
use uvvm_vvc_framework.ti_data_fifo_pkg.all;

library vip_avalon_st;
use vip_avalon_st.avalon_st_bfm_pkg.all;
use vip_avalon_st.vvc_methods_pkg.all;
use vip_avalon_st.td_vvc_framework_common_methods_pkg.all;


-- Test bench entity
entity tb_avalon_st_vvc is
end entity;

-- Test bench architecture
architecture func of tb_avalon_st_vvc is
   constant C_SCOPE              : string  := C_TB_SCOPE_DEFAULT;

   -- Clock and bit period settings
   constant C_CLK_PERIOD         : time := 10 ns;
   constant C_BIT_PERIOD         : time := 16 * C_CLK_PERIOD;

   -- Avalon-ST bus widths
   --constant C_CHANNEL_WIDTH   : natural := 1;
   constant C_DATA_WIDTH      : natural := 64;
   constant C_BITS_PER_SYMBOL : natural := 8;
   constant C_DATA_LENGTH     : natural := 512;
   --constant C_ERROR_WIDTH     : natural := 1;
   constant C_EMPTY_WIDTH     : natural := 4;
begin
   -----------------------------------------------------------------------------
   -- Instantiate test harness, containing DUT and Executors
   -----------------------------------------------------------------------------
   i_test_harness : entity work.th_avalon_st_vvc
   generic map(
         --CHANNEL_WIDTH      => C_CHANNEL_WIDTH,
         DATA_WIDTH         => C_DATA_WIDTH,
         --ERROR_WIDTH        => C_ERROR_WIDTH,
         EMPTY_WIDTH        => C_EMPTY_WIDTH
      );

   ------------------------------------------------
   -- PROCESS: p_main
   ------------------------------------------------
   p_main: process
      variable v_data_array    : t_slv_array(0 to C_DATA_LENGTH-1)(C_DATA_WIDTH-1 downto 0) := (others => (others =>
         ↪    '0'));
      variable v_empty         : std_logic_vector(C_EMPTY_WIDTH-1 downto 0) := (others => '0');

      variable v_num_test_loops  : natural := 0;
   begin

   -- Wait for UVVM to finish initialization
   await_uvvm_initialization(VOID);

   -- Print the configuration to the log
   report_global_ctrl(VOID);
   report_msg_id_panel(VOID);

   -----------------------------------------------------------------------------
   -- Enable log message
   -----------------------------------------------------------------------------
   disable_log_msg(ALL_MESSAGES);
   --enable_log_msg(ALL_MESSAGES);
   enable_log_msg(ID_LOG_HDR);
   --enable_log_msg(ID_DATA);
   --enable_log_msg(ID_SEQUENCER);
```

```
86      --enable_log_msg(ID_UVVM_SEND_CMD);
87
88      disable_log_msg(AVALON_ST_VVCT, 1, TX, ALL_MESSAGES);
89      disable_log_msg(AVALON_ST_VVCT, 1, RX, ALL_MESSAGES);
90      --enable_log_msg(AVALON_ST_VVCT, 1, TX, ALL_MESSAGES);
91      --enable_log_msg(AVALON_ST_VVCT, 1, RX, ALL_MESSAGES);
92
93      enable_log_msg(AVALON_ST_VVCT, 1, TX, ID_PACKET_INITIATE);
94      enable_log_msg(AVALON_ST_VVCT, 1, TX, ID_PACKET_COMPLETE);
95      enable_log_msg(AVALON_ST_VVCT, 1, RX, ID_PACKET_INITIATE);
96      enable_log_msg(AVALON_ST_VVCT, 1, RX, ID_PACKET_COMPLETE);
97
98      ----------------------------------------------------------------------------
99      -- Enable/disable Avalon-ST signals
100     ----------------------------------------------------------------------------
101     shared_avalon_st_vvc_config(TX, 1).bfm_config.use_channel   := false;
102     shared_avalon_st_vvc_config(TX, 1).bfm_config.use_error     := false;
103     shared_avalon_st_vvc_config(TX, 1).bfm_config.use_empty     := true;
104
105     -- Percent of cycles the receive module should assert ready_o signal
106     shared_avalon_st_vvc_config(RX, 1).bfm_config.ready_percentage := 100;
107
108     -- Set empty signal if some symbols are empty at the last transmission
109     v_empty := std_logic_vector(to_unsigned(0, v_empty'length));
110
111
112     log(ID_LOG_HDR, "Starting simulation of TB scaler vvc", C_SCOPE);
113     log("Wait 10 clock period for reset to be turned off");
114     wait for (10 * C_CLK_PERIOD);
115
116     -- Number of times to run the test loop
117     v_num_test_loops := 50;
118
119     for i in 1 to v_num_test_loops loop
120        -- Create a random ready percentage for the recieve module
121        shared_avalon_st_vvc_config(RX, 1).bfm_config.ready_percentage := random(1,100);
122
123        -- Random empty signal between 0 and number of symbols - 1.
124        v_empty := std_logic_vector(to_unsigned(random(0,(C_DATA_WIDTH/C_BITS_PER_SYMBOL)-1), v_empty'length));
125
126        -- Write random data to data_array
127        for j in v_data_array'range loop
128           -- Generate random data
129           v_data_array(j) := random(C_DATA_WIDTH);
130        end loop;
131
132        -- Margin
133        wait for 10*C_CLK_PERIOD;
134
135        -- Start send and receive VVC
136        avalon_st_send(AVALON_ST_VVCT, 1, v_data_array, v_empty, "Sending v_data_array");
137        avalon_st_expect(AVALON_ST_VVCT, 1, v_data_array, v_empty, "Checking data", ERROR);
138     end loop;
139
140     -- Wait for completion
141     await_completion(AVALON_ST_VVCT, 1, RX, 100*C_DATA_LENGTH*v_num_test_loops*C_CLK_PERIOD);
142
143
144     log(ID_LOG_HDR, "Completion of avalon_st_reviece", C_SCOPE);
145
146     ----------------------------------------------------------------------------
147     -- Ending the simulation
148     ----------------------------------------------------------------------------
149     wait for 1000 ns;             -- to allow some time for completion
150     report_alert_counters(FINAL); -- Report final counters and print conclusion for simulation (Success/Fail)
151     log(ID_LOG_HDR, "SIMULATION COMPLETED", C_SCOPE);
152
153     -- Finish the simulation
154     std.env.stop;
155     wait;  -- to stop completely
156
157     end process p_main;
158
159  end func;
```

C.3 Avalon-ST VVC Testharness

```
1  --------------------------------------------------------------------------------------------
2  -- Project: FPGA video scaler
3  -- Author: Thomas Stenseth
4  -- Date: 2019-03-11
5  -- Version: 0.1
6  --------------------------------------------------------------------------------------------
7  -- Description:
8  --------------------------------------------------------------------------------------------
9  -- v0.1:
10 --------------------------------------------------------------------------------------------
11
12
13 library IEEE;
14 use IEEE.std_logic_1164.all;
15 use IEEE.numeric_std.all;
16
17 library uvvm_util;
18 context uvvm_util.uvvm_util_context;
19
20 library uvvm_vvc_framework;
21 use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;
22 use uvvm_vvc_framework.ti_data_fifo_pkg.all;
23
24 library vip_avalon_st;
25
26
27 -- Test harness entity
28 entity th_avalon_st_vvc is
29    generic (
30          --CHANNEL_WIDTH      : natural;
31          DATA_WIDTH        : natural;
32          --ERROR_WIDTH       : natural;
33          EMPTY_WIDTH       : natural
34       );
35 end entity;
36
37 -- Test harness architecture
38 architecture struct of th_avalon_st_vvc is
39    -- DSP interface and general control signals
40    signal clk_i                : std_logic := '0';
41    signal sreset_i             : std_logic := '0';
42
43    -- Sink
44    --signal channel_i            : std_logic_vector(CHANNEL_WIDTH - 1 downto 0);
45    signal data_i               : std_logic_vector(DATA_WIDTH - 1 downto 0);
46    --signal error_i              : std_logic_vector(ERROR_WIDTH - 1 downto 0);
47    signal ready_o              : std_logic;
48    signal valid_i              : std_logic := '0';
49    signal empty_i              : std_logic_vector(EMPTY_WIDTH - 1 downto 0);
50    signal endofpacket_i        : std_logic := '0';
51    signal startofpacket_i      : std_logic := '0';
52    signal check_data           : std_logic_vector(DATA_WIDTH -1 downto 0);
53
54    -- Source
55    --signal channel_o            : std_logic_vector(CHANNEL_WIDTH - 1 downto 0);
56    signal data_o               : std_logic_vector(DATA_WIDTH - 1 downto 0);
57    --signal error_o              : std_logic_vector(ERROR_WIDTH - 1 downto 0);
58    signal ready_i              : std_logic := '0';
59    signal valid_o              : std_logic;
60    signal empty_o              : std_logic_vector(EMPTY_WIDTH - 1 downto 0);
61    signal endofpacket_o        : std_logic;
62    signal startofpacket_o      : std_logic;
63
64
65    constant C_CLK_PERIOD : time := 10 ns; -- 100 MHz
66 begin
67    ----------------------------------------------------------------------------
68    -- Instantiate the concurrent procedure that initializes UVVM
69    ----------------------------------------------------------------------------
70    i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;
71
72    ----------------------------------------------------------------------------
73    -- AVALON ST VVC
74    ----------------------------------------------------------------------------
75    i1_avalon_st_vvc: entity vip_avalon_st.avalon_st_vvc
76    generic map(
77       --GC_CHANNEL_WIDTH  => CHANNEL_WIDTH,
78       GC_DATA_WIDTH        => DATA_WIDTH,
79       --GC_ERROR_WIDTH    => ERROR_WIDTH,
80       GC_EMPTY_WIDTH       => EMPTY_WIDTH,
81       GC_INSTANCE_IDX      => 1
82    )
83    port map(
84       clk   => clk_i,
85       -- Sink
86       --avalon_st_sink_if.channel_i            => channel_i,
```

```
87          avalon_st_sink_if.data_i               => data_i,
88          --avalon_st_sink_if.error_i              => error_i,
89          avalon_st_sink_if.ready_o              => ready_o,
90          avalon_st_sink_if.valid_i              => valid_i,
91          avalon_st_sink_if.empty_i              => empty_i,
92          avalon_st_sink_if.endofpacket_i        => endofpacket_i,
93          avalon_st_sink_if.startofpacket_i      => startofpacket_i,
94
95          -- Source
96          --avalon_st_source_if.channel_o          => channel_o,
97          avalon_st_source_if.data_o             => data_o,
98          --avalon_st_source_if.error_o            => error_o,
99          avalon_st_source_if.ready_i            => ready_i,
100         avalon_st_source_if.valid_o            => valid_o,
101         avalon_st_source_if.empty_o            => empty_o,
102         avalon_st_source_if.endofpacket_o      => endofpacket_o,
103         avalon_st_source_if.startofpacket_o    => startofpacket_o
104      );
105
106      data_i <= data_o;
107      ready_i <= ready_o;
108      valid_i <= valid_o;
109      empty_i <= empty_o;
110      startofpacket_i <= startofpacket_o;
111      endofpacket_i <= endofpacket_o;
112
113
114      ----------------------------------------------------------------------------
115      -- Reset process
116      ----------------------------------------------------------------------------
117      -- Toggle the reset after 5 clock periods
118      p_sreset: sreset_i <= '1', '0' after 5 *C_CLK_PERIOD;
119
120      ----------------------------------------------------------------------------
121      -- Clock process
122      ----------------------------------------------------------------------------
123      p_clk: process
124      begin
125         clk_i <= '0', '1' after C_CLK_PERIOD / 2;
126         wait for C_CLK_PERIOD;
127      end process;
128
129   end struct;
```

Appendix D

MATLAB source code

D.1 Image to Binary Function

```
%% Converts image to binary serial data

function output = img2bin(img, filename, bits)
    fileID = fopen(filename,'w');
    [height,width,colors] = size(img);
    for y = 1:height
        for x = 1:width
            for z = colors:-1:1
                fprintf(fileID,dec2bin(img(y,x,z),bits));
            end
            if (y == height) && (x == width) && (z == 1)
                % No new line
            else
                fprintf(fileID,'\n');
            end
        end
    end
    fclose(fileID);
    output = img;
end
```

D.2 Binary to Image Function

```
%% Converts binary serial data to image

function output = bin2img(filename, width, height, colours, bits)
    % Read data from file
    fileID = fopen(filename);
    num = (width*height*(bits*3+2) - 2);
    data_serial = fread(fileID,[1 num], '*char');

    % Convert data to characters
    data_lines = splitlines(data_serial);

    if colours
        for i = 1:size(data_lines)
            colour_a(i,1) = extractBetween(data_lines(i), bits*2 + 1, bits*3);
            colour_b(i,1) = extractBetween(data_lines(i), bits + 1, bits*2);
            colour_c(i,1) = extractBetween(data_lines(i), 1, bits);
        end

        data_a = bin2dec(colour_a);
        data_b = bin2dec(colour_b);
        data_c = bin2dec(colour_c);
    else
        data = bin2dec(data_lines);
    end

    for y = 1:height
        for x = 1:width
            if colours
                img(y,x,1) = data_a(((y-1)*width)+x);
                img(y,x,2) = data_b(((y-1)*width)+x);
                img(y,x,3) = data_c(((y-1)*width)+x);
            else
                img(y,x) = data(((y-1)*width)+x);
            end
        end
    end
    fclose(fileID);
    output = uint8(img);
end
```

Appendix E

Complete test results

E.1 Nearest-Neighbor Interpolation

Test image	Source	PSNR	MSE	SSIM
Lion King	360p	33.5059	29.0063	0.9913
Lion King	540p	37.5863	11.3357	0.9970
Lion King	720p	36.8666	13.3790	0.9961
Toy Story	360p	28.4989	91.8733	0.9326
Toy Story	540p	32.0706	40.3661	0.9718
Toy Story	720p	32.0858	40.2250	0.9737

Table E.1: MATLAB nearest-neighbor, animated content

Test image	Source	PSNR	MSE	SSIM
Lion King	360p	33.4575	29.3309	0.9911
Lion King	540p	37.4891	11.5924	0.9969
Lion King	720p	35.7447	17.3224	0.9954
Toy Story	360p	28.4814	92.2444	0.9319
Toy Story	540p	32.0355	40.6944	0.9712
Toy Story	720p	30.5699	57.0289	0.9655

Table E.2: VHDL nearest-neighbor, animated content

Test image	Source	PSNR	MSE	SSIM
Jaguar	360p	36.5703	14.3235	0.9702
Jaguar	540p	39.9563	6.5682	0.9859
Jaguar	720p	40.3204	6.0401	0.9877
Lemur	360p	38.7634	8.6446	0.9706
Lemur	540p	41.8370	4.2597	0.9856
Lemur	720p	42.3832	3.7563	0.9881
Birds	360p	34.0866	25.3759	0.9305
Birds	540p	37.4493	11.6991	0.9672
Birds	720p	37.7155	11.0036	0.9713

Table E.3: MATLAB nearest-neighbor, natural content

Test image	Source	PSNR	MSE	SSIM
Jaguar	360p	36.4504	14.7246	0.9693
Jaguar	540p	39.7004	6.9669	0.9849
Jaguar	720p	38.6753	8.8217	0.9834
Lemur	360p	38.5650	9.0486	0.9693
Lemur	540p	41.4439	4.6633	0.9842
Lemur	720p	40.6418	5.6092	0.9844
Birds	360p	34.0191	25.7733	0.9286
Birds	540p	37.3036	12.0981	0.9652
Birds	720p	36.3976	14.9046	0.9626

Table E.4: VHDL nearest-neighbor, natural content

E.2 Bilinear Interpolation

Test image	Source	PSNR	MSE	SSIM
Lion King	360p	35.1059	20.0675	0.9942
Lion King	540p	37.8885	10.5739	0.9968
Lion King	720p	40.3181	6.0433	0.9980
Toy Story	360p	29.6236	70.9120	0.9444
Toy Story	540p	32.7453	34.5584	0.9728
Toy Story	720p	35.8528	16.8967	0.9863

Table E.5: MATLAB bilinear, animated content

Test image	Source	PSNR	MSE	SSIM
Lion King	360p	31.4662	46.3935	0.9910
Lion King	540p	32.6180	35.5860	0.9939
Lion King	720p	38.7839	8.6039	0.9973
Toy Story	360p	28.1786	98.9051	0.9373
Toy Story	540p	30.4567	58.5343	0.9678
Toy Story	720p	34.4312	23.4402	0.9821

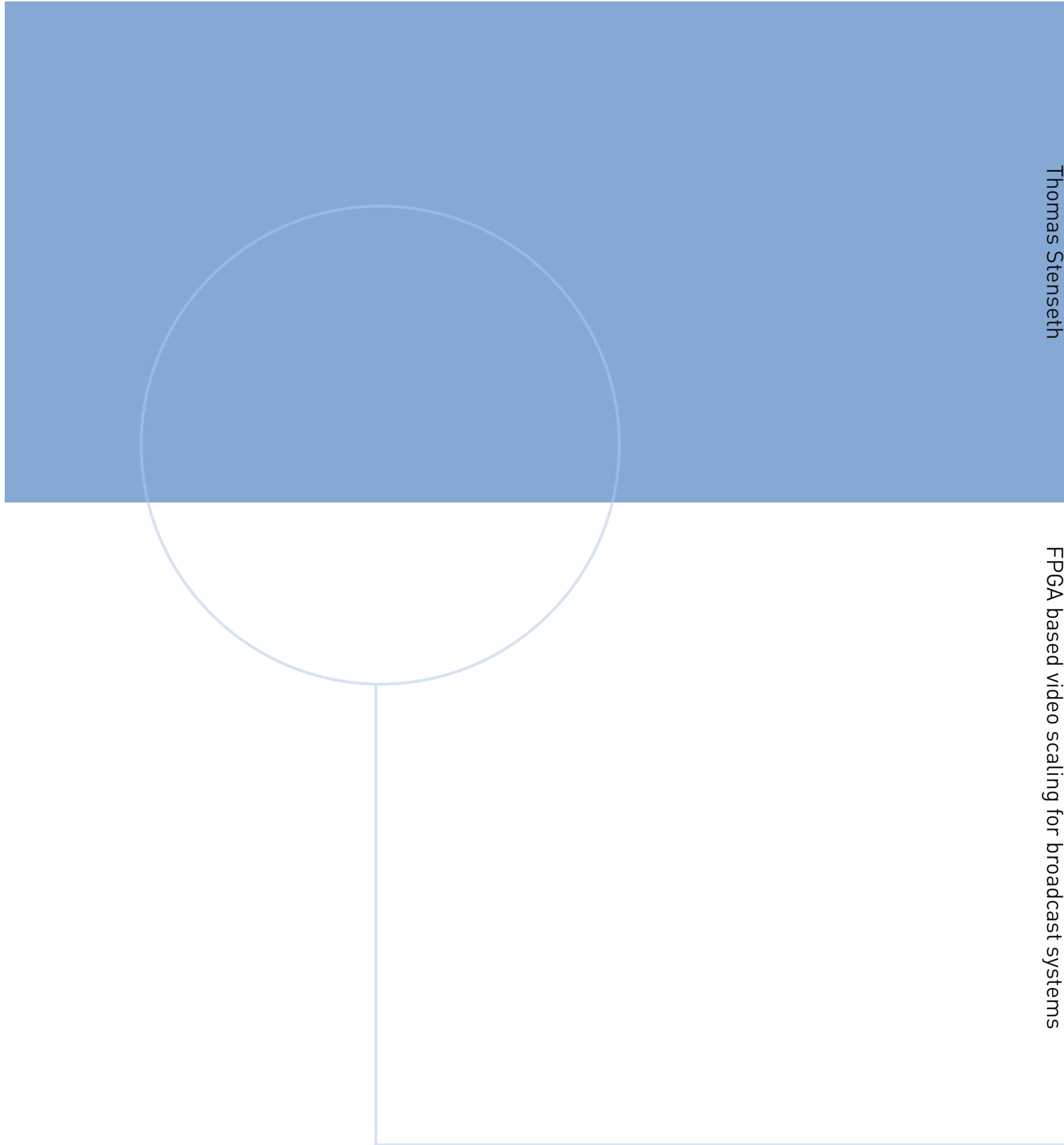
Table E.6: VHDL bilinear, animated content

Test image	Source	PSNR	MSE	SSIM
Jaguar	360p	39.1964	7.8242	0.9822
Jaguar	540p	41.9287	4.1707	0.9900
Jaguar	720p	44.6456	2.2311	0.9945
Lemur	360p	39.8598	6.7159	0.9768
Lemur	540p	42.7464	3.4550	0.9879
Lemur	720p	45.9900	1.6371	0.9942
Birds	360p	35.7531	17.2892	0.9472
Birds	540p	39.0925	8.0136	0.9739
Birds	720p	42.5781	3.5915	0.9876

Table E.7: MATLAB bilinear, natural content

Test image	Source	PSNR	MSE	SSIM
Jaguar	360p	32.2711	38.5453	0.9755
Jaguar	540p	32.8332	33.8656	0.9844
Jaguar	720p	41.3190	4.7993	0.9911
Lemur	360p	34.9387	20.8551	0.9698
Lemur	540p	35.8418	16.9394	0.9818
Lemur	720p	42.7182	3.4774	0.9906
Birds	360p	31.3181	48.0027	0.9368
Birds	540p	32.4206	37.2410	0.9658
Birds	720p	39.7680	6.8593	0.9802

Table E.8: VHDL bilinear, natural content



NTNU
Norwegian University of
Science and Technology