Norwegian University of
Science and Technology

FPGA Implementation of a Video Scaler

Roger Skarbø

Master of Science in Electronics
Submission date: June 2010
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Jon Erik Oterhals, ARM

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

FPGA implementation of a video scaler.

Video scaling is commonly used to convert video source material to the native resolution of the display device, e.g. in a TV. The resulting image quality is highly dependant on the algorithm used to scale the image.

An efficient HW scaler can ease the job of the GPU (Graphic Processing Unit), by providing the video stream in a resolution close to the wanted texture resolution. In addition to up- or downscaling, 90 degree rotation, and format conversion are useful properties of a video scaler.

The task is:

- Analysis of different video scaling algorithms with respect to resulting quality, up-, down-scaling, rotation, conversion, and also upscaling of computer generated graphics.

- Choosing a set of algorithms for implementation on an FPGA system.

- Qualifying the performance and quality of the implemented version(s) of the algorithm, and comparing this with known video scalers.

Assignment given: 15. January 2010
Supervisor: Kjetil Svarstad, IET

Summary

Three algorithms for video scaling were developed and tested in software, for implementation on an FPGA. Two of the algorithms were implemented in a video scaler system. These two algorithms scale up with factors 1.25 and 1.875, which is used for scaling SD WIDE to HD resolution and SD WIDE to FullHD resolution, respectively. An algorithm with scaling factor 1.5, scaling HD to FullHD, was also discussed, but not implemented.

The video scaler was tested with a verilog testbench provided by ARM. When passing the testbench, the video scaler system was loaded on an FPGA. Results from the FPGA were compared with the software algorithms and the simulation results from the testbench. The video scaler implemented on the FPGA produced predictable results.

Even though a fully functional video scaler was made, there were not time left to create the necessary software drivers and application software that would be needed to run the video scaler in real time with live video output. So a comparison of the output from the implemented algorithms is performed with common scaling algorithms used in video scalers, such as bilinear interpolation and bicubic interpolation.

This thesis also deal with graphics scaling. Some well-known algorithms for graphic scaling were written in software, including a self-made algorithm to suit hardware. These algorithms were not implemented in hardware, but comparison of the results are performed.

Preface

This Master thesis has been performed in the spring of 2010 at ARM Trondheim. ARM presented the project *FPGA implementation of a video scaler* that was assigned to me.

I want to give a special thanks to my two supervisors at ARM and NTNU, Jon Erik Oterhals and Kjetil Svarstad, respectively. Both supervisors answered all my questions willingly when help was needed. Also, I want to thank ARM Trondheim, for the opportunity to work with future-oriented architectures, and all the ARM employees that has been helping me.

June 22, 2010

Roger Skarbø

Contents

1 Introduction 1
1.1 Narrowing the assignment . . . 1
1.2 Research process . . . 1
1.3 Contributions . . . 2

2 Theory on video scaling 3
2.1 Pixel formats . . . 3
2.2 Bilinear interpolation . . . 4
2.3 Video scaling algorithms used for comparison . . . 5
2.4 SD WIDE . . . 5

3 Scaling algorithms 6
3.1 Graphic scaling, RGB . . . 6
3.1.1 Scale2x algorithm . . . 6
3.1.2 Scale3x algorithm . . . 7
3.1.3 Hq3x . . . 9
3.1.4 Rogers2x algorithm for hardware . . . 9
3.2 Video scaling algorithms for hardware . . . 12

4 AMBA (Advance Microcontroller Bus Architecture) 16
4.1 AMBA APB (Advanced Peripheral Bus) . . . 16
4.1.1 APB module . . . 16
4.2 AMBA AXI (Advance eXtensible Interface) . . . 19
4.2.1 AXI Read module . . . 19
4.2.2 AXI Write module . . . 20

5 Verilog testbench 22

6 Implementation in hardware 24
6.1 Testing AXI and APB . . . 24
6.2 Memory control module (mem_reg) . . . 25
6.3 Scale 1.25 (SD WIDE to HD) . . . 29
6.4 Scale 1.875 (SD WIDE to FullHD) . . . 33
6.5 Test of video scaler system . . . 37
6.5.1 Implementing system on the FPGA . . . 38

7 Results 40
7.1 Performance in FPGA . . . 40
7.2 Picture quality . . . 41
7.2.1 Scale factor 1.875 . . . 41
7.2.2 Scale factor 1.25 . . . 43
7.2.3 Scale factor 1.5 . . . 43

8 Discussion **45**
8.1 Scale 1.5 (HD2FullHD) 45
8.2 Smoothing effect on SD2FullHD 46
8.3 Similarities with bilinear interpolation 48
8.4 Truncation on the algorithms 50
8.5 Performance of implemented modules 50
8.6 Merging the video scaler systems 52

9 Conclusion **53**

10 Future work **54**

A C++ code **57**
A.1 Algorithms for SD/HD/FullHD up-scaling 57
A.2 Algorithms for graphic scaling (RGB) 65
A.3 Rogers2x algorithm 78

B Verilog code **85**
B.1 AXI Read module 85
B.2 AXI Write module 89
B.3 APB module 95
B.4 Memory Control module (mem_reg) 99
B.5 Scale 1.25 module 104
B.6 Scale 1.875 module 119
B.7 RAM module 134

C Video scaler system **135**

D Log files **137**
D.1 Video scaler 1.25 - Xilinx log file 137
D.2 Video scaler 1.875 - Xilinx log file 140
D.3 Scale 1.25 module - Synplify log file 143
D.4 Scale 1.875 module - Synplify log file 145
D.5 Mem_reg module - Synplify log file 147

List of Figures

1 Bilinear interpolation, [2]. 4
2 Scale2x algorithm. Shows expansion of one input pixel. 7
3 Original picture (zoom 200 %) and Scale2x effect. 7
4 Scale3x algorithm. Shows expansion of one input pixel. 8
5 Original picture (zoom 300 %) and Scale3x effect. 8
6 Original picture (zoom 300 %) and hq3x effect. 9
7 Illustration colour distortion of Rogers2x algorithm. Original picture (left, zoom 200 %) and Rogers2x algorithm (right). 10
8 Rogers2x algorithm. Shows expansion of one input pixel. 11
9 Original picture (zoom 200 %), Scale2x algorithm, and Rogers2x algorithm. 11
10 Output dependencies with linear scaling, scaling factor 1.25. 12
11 Output dependencies with linear scaling, scaling factor 1.5. 13
12 Output dependencies with linear scaling, scaling factor 1.875. 14
13 AXI read overlapping burst example (burst of 8 transfers). 20
14 AXI write burst example (burst of 8 transfers). 21
15 Verilog toplevel testbench structure. 22
16 Test of AXI and APB interface. 24
17 Memory management: Reading from RAM once. 27
18 Output of scaling factor 1.25 with one RAM read. 28
19 Output of scaling factor 1.875 with one RAM read. 29
20 Memory management: Reading from RAM twice. 30
21 Internal register representation of Scale 1.25 module and output. 31
22 Internal register representation of Scale 1.875 module and output. 36
23 Video scaler system. 37
24 Overview of the system implemented on the FPGA. 38
25 SD2FullHD accurate to dependencies (400% zoom). 47
26 SD2FullHD with smoothing effect (400% zoom). 47
27 GIMP Linear output (400% zoom). 47
28 Original picture of Lenna's eye (400% zoom). 47
29 Part of the ASCII characters, source picture. 48
30 ASCII characters scaled up with a factor of 1.875, bilinear interpolation (top) and SD2FullHD (bottom). 49
31 ASCII characters scaled up with a factor of 1.5, bilinear interpolation (top) and SD2FullHD (bottom). 49
32 ASCII characters scaled up with a factor of 1.25, bilinear interpolation (top) and SD2FullHD (bottom). 50

List of Tables

1 APB - Signal connected to the video scaler 18
2 Video scaler performance in FPGA with scaling factor 1.25. 40
3 Video scaler performance in FPGA with scaling factor 1.875. 41
4 Comparing FPGA implementation (SD2FullHD), with scaling factor 1.875, with GIMP scaling algorithms. 42
5 Comparing FPGA implementation (SD2FullHD), with scale factor 1.875, with Corel scaling algorithms. 42
6 Comparing FPGA implementation, with scale factor 1.25, with GIMP scaling algorithms. 43
7 Comparing FPGA implementation, with scale factor 1.5, with GIMP scaling algorithms. 43
8 Effect of lacking smoothness. 47
9 Effect of truncation versus rounding up. 51
10 Scale 1.25 module performance. 51
11 Scale 1.875 module performance. 51
12 Mem_reg module performance. 52

1 Introduction

A video scaler for implementation on an FPGA will be specified and implemented. An analysis of different video scaling algorithms and computer generated graphics scaling with respect to resulting quality should be performed. A set of algorithms is implemented on an FPGA system. When the algorithms have been implemented on an FPGA, performance and quality should be compared with known video scalers.

1.1 Narrowing the assignment

The project description describes an assignment to make a fully functional video scaler, this assignment was too large with regards to the time constraints on this project. When making a video scaler in hardware, oppose to flexible software, there are a lot of constraints. After consulting with the supervisor at ARM, the assignment was narrowed to scaling up from SD WIDE resolution (1024x576) to HD resolution (1280x720) and FullHD resolution (1920x1080). So the part in the project description that included downscaling and rotation was dropped.

Even though a fully functional video scaler was made, there were not time left to create the necessary software drivers and application software that would be needed to run the video scaler in real time with live video output. So comparison is performed with common scaling algorithms used in video scalers, such as bilinear interpolation and bicubic interpolation.

The project description concerning up-scaling of computer generated graphics was not changed.

1.2 Research process

The research process started with an introduction of scaling in both video and graphics. This included an understanding of the pixel formats RGB and YC_bC_r. Different algorithms were analysed before choosing a set of algorithms to implement on an FPGA. Qualifying the performance and quality of the algorithms that were relevant for implementation were carried out throughout the whole research process.

The project was started on the 11. of January 2010, and access to the lab at ARM Trondheim was given the 12. of April. Then started the process of implementing the algorithms on the FPGA.

1.3 Contributions

A set of algorithms for video scaling were developed in software with great concern to how it easily could be implemented in hardware, they are referred to as *SD2HD*, *SD2FullHD* and *HD2FullHD* in the text. These algorithms were then converted to synthesizable Verilog code and implemented in an FPGA. In addition, a set of algorithms for graphics scaling were written in software, but not tested in a hardware solution. An algorithm for graphics scaling was developed to suit a hardware implementation, called *rogers2xAvgGFX*.

The assignment also included thorough introduction to the AMBA system ARM has developed. To test any thing on the FPGA, the AMBA modules used in the FPGA had to be implemented correctly. This became a large part of the project. Three modules were written, AXI Read module, AXI Write module and APB module, all modules were written with regards of the AMBA specifications in [1].

2 Theory on video scaling

This section include some theory used in the thesis. RGB and YUV pixel formats is first described, then common algorithms for video scaling is described. Bilinear interpolation is described more in details because the implemented algorithms are similar the this interpolation. Lastly, a short description of SD WIDE resolution is given.

2.1 Pixel formats

There are two primary colour spaces to digital present a pixel values, these are the RGB pixel format and the YC_bC_r pixel format (referred to as YUV in the text). The RGB pixel format has three components, one red, one green and one blue. The difference between YC_bC_r and RGB is that YC_bC_r represent colour as brightness, Y (luma), and two colour difference signals, C_b (B-Y) and C_r (R-Y). B-Y stands for blue minus luma, and R-Y stands for red minus luma. The RGB pixel format is usually used for computer graphics, and YUV for video.

[12] describes in chapter 5.5 how the human eye respond to the different wavelength of red, green and blue. The highest response of the eye is with the green colour, followed by red, and then blue. Both [12] and [7] take these dependencies in to account when doing RGB to YUV conversion:

$$\begin{aligned} \mathrm{Y} &= 0.299\mathrm{R} + 0.587\mathrm{G} + 0.114\mathrm{B} \\ \mathrm{C}_r = \mathrm{V'} &= (\mathrm{R\text{-}Y})*0.713 \\ \mathrm{C}_b = \mathrm{U'} &= (\mathrm{B\text{-}Y})*0.565 \end{aligned}$$

To convert from YUV to RGB the following formula is used:

$$\begin{aligned} \mathrm{R} &= \mathrm{Y} + 1.403\mathrm{V'} \\ \mathrm{G} &= \mathrm{Y} - 0.344\mathrm{U'} - 0.714\mathrm{V'} \\ \mathrm{B} &= \mathrm{Y} + 1.770\mathrm{U'} \end{aligned}$$

The pixels can be represented in a different formats, for a full overview of the different formats see [19]. In this project, RGB format is represented as 24 bit large data packets, which contain 8 bit large red, green and blue values. For YUV, it is either represented in a packed format, or a planar format. In the packed format, the Y, U and V are packed together in to macro pixels which are stored in a single array. The planar format store each component as separate array. For instance, yuv444 format sends all 8 bit Y components first, followed by 8 bit U plane, and then the 8 bit V plane (24 bits per pixel). The main reason for using YUV format is to reduce the bits per pixel, and one of the most used planar format in video codecs is the yuv420 format (referred to as YV12 in [19]). This planar format first send 8 bit Y plane followed by 8 bit 2x2 sub-sampled V and U planes, giving an average of 12 bit per pixel.

2.2 Bilinear interpolation

Bilinear interpolation [2] considers the four closest input pixels surrounding a unknown pixel. Linear interpolation is performed in both directions, vertical and horizontal, hence the name bilinear. The result is independent on the order of the interpolation. Bilinear interpolation produces a weighted average of the four input pixels to calculate the output value.

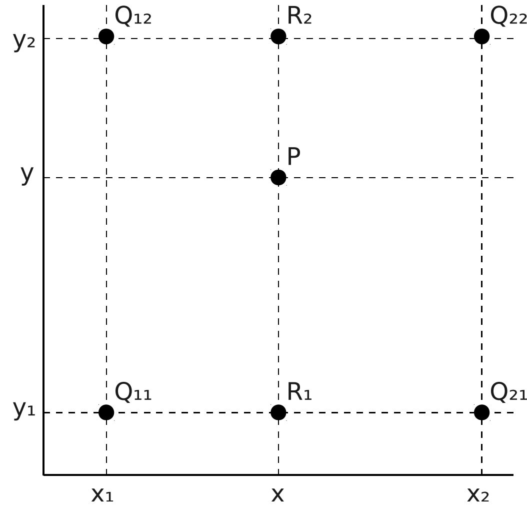


Figure 1: Bilinear interpolation, [2].

If we assume that the four surrounding input pixel points are known, the value of an unknown function f of output pixel $P = (x, y)$ could be found. The four input pixel points are represented as $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$. Figure 1 show an illustration of how P is calculated depending on the four input pixels.

First we do linear interpolation in the x-direction.

$$f(R_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{1}$$

where $R_1 = (x, y_1)$.

$$f(R_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \tag{2}$$

where $R_2 = (x, y_2)$.

Secondly, we proceed by interpolating in the y-direction.

$$f(P) = \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2) \tag{3}$$

Equation 4 shows the full extent of function $f(x, y)$.

$$\begin{aligned} f(x,y) = &\frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y) \\ +&\frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y_2 - y) \\ +&\frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y - y_1) \\ +&\frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y - y_1) \end{aligned} \tag{4}$$

By choosing a coordinate system which the four input pixel points $f(0, 0)$, $f(0, 1)$, $f(1, 0)$ and $f(1, 1)$ are known, the interpolation formula simplifies to

$$f(x, y) = f(0, 0)\,(1 - x)(1 - y) + f(1, 0)\,x(1 - y) + f(0, 1)\,(1 - x)y + f(1, 1) \tag{5}$$

It is well know in the image scaling environment that the bilinear interpolation causes some undesirable softening of details and can still be somewhat jagged because it only depend on four input pixels. Nvidia [9] gives a more thorough discussion on this topic.

2.3 Video scaling algorithms used for comparison

The implemented scaling algorithms is compared with well-known scaling algorithms. The comparison is with bilinear interpolation, described in the previous sub-section, bicubic interpolation and Lanczos3 window filter.

The bicubic interpolation use sixteen surrounding input pixels to calculate a new pixel. Since bicubic use sixteen pixels, in contrast to the four bilinear uses, the bicubic interpolation is seen upon to be much better than bilinear interpolation.

The Lanczos3 window filter [3] use two sinc functions to do interpolation. Therefore, it is also called ”Sinc” window. The Lanczos3 is a common method used when performing video scaling.

2.4 SD WIDE

In PAL SD with 16:9 format is encoded to 720x576 pixels, with a Pixel Aspect Ratio of 64:45 (Anamorphic). For displays that uses square pixels (aspect ratio 1:1) this resolution is converted to 1024x576 pixels (16:9).

When referring to SD WIDE in the text, the resolution is 1024x576.

3 Scaling algorithms

This section explain how the different scaling algorithms were tested. First a description of the graphics scaling algorithms are given, then follows a description of the process of developing video scaling algorithms with three static scaling factors.

To begin testing the different scaling algorithms it was necessary to use a C++ library suitable for pictures. Easy BMP [6] was chosen as the suitable library. This library is designed for easily reading, writing, and modifying Windows bitmap (BMP) image files. This library had a resizing option by using bilinear interpolation. This bilinear interpolation gave poor results. It was most likely implemented incorrectly. A new method of doing interpolation was introduced, this method can be found in section 3.2.

Only few parts of the Easy BMP library were used, the rest was striped. The parts of reading the header information of the bmp file and the making of the output header, and a few Set/Tell functions were used. These functions would not be difficult to make, but by using parts of this library time could be saved.

Now that the foundation was established, the algorithms could be imported by including a self-made ”Algorithms.h” file in the main program. This header file include all the algorithms that were to be tested.

3.1 Graphic scaling, RGB

Several graphic scaling algorithms were tested in software to compare the scaling results. Some of the ones that were tested can be found in [14], this included Nearest neighbour algorithm, Scale2x algorithm, Scale3x algorithm and hq3x algorithm. These algorithms goes under the category pixel art scaling algorithms.

The Nearest neighbour algorithm [17] produced, as expected, poor results. This algorithm produce output pixels by duplicating the nearest input pixel, and takes no concern of the surrounding pixels. This algorithm was only used to get started using the Easy BMP library.

3.1.1 Scale2x algorithm

The Scale2x algorithm was the next to be tested. A description of the algorithm is found in [15], and the code used for scaling is in appendix A.2 under the function name **scale2xAlgorithm**. The Scale2x algorithm uses the input pixels above, to both the sides and the pixel under the input pixel that is going to be expanded. In figure 2 it shows how input pixel E is expanded to $E0$, $E1$, $E2$, and $E3$.

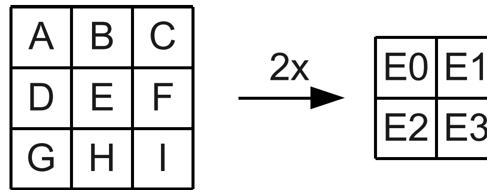


Figure 2: Scale2x algorithm. Shows expansion of one input pixel.

Scale2x use a set of rules, these are (described in C):

```
E0 = D == B && B != F && D != H ? D : E;
E1 = B == F && B != D && F != H ? F : E;
E2 = D == H && D != B && H != F ? D : E;
E3 = H == F && D != H && B != F ? F : E;
```

An illustration of how Scale2x algorithm works is shown in figure 3.

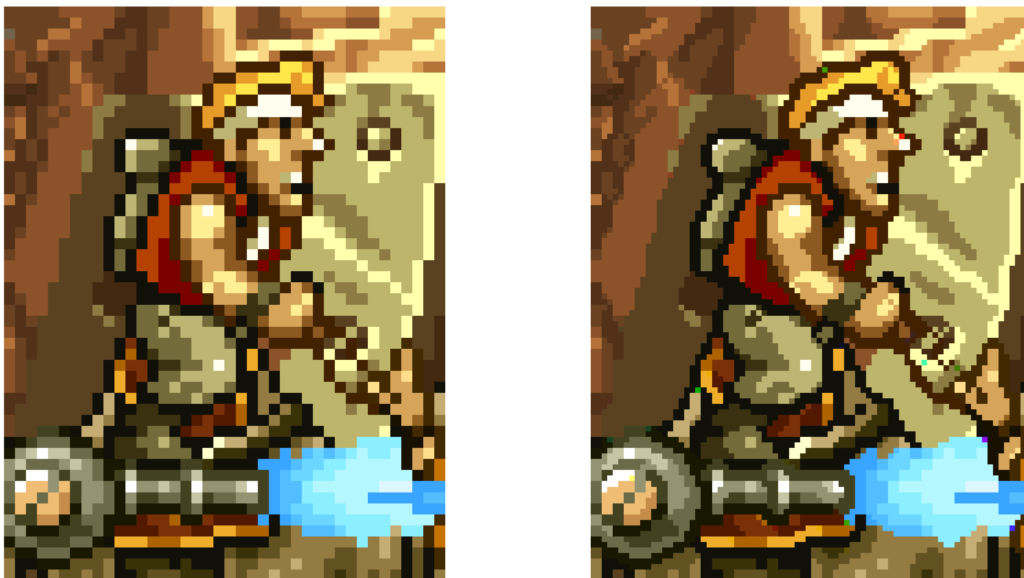


Figure 3: Original picture (zoom 200 %) and Scale2x effect.

3.1.2 Scale3x algorithm

The Scale3x algorithm was the next to be tested. A description of the algorithm is found in [15], and the code used for scaling is in appendix A.2 under the function name **scale3xAlgorithm**. The Scale3x algorithm uses all the surrounding input pixels to expand one pixel. In figure 4 it shows how input pixel E is expanded to nine different pixels.

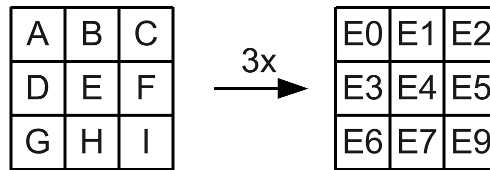


Figure 4: Scale3x algorithm. Shows expansion of one input pixel.

Scale3x use a set of rules, these are (described in C):

```
E0 =  D == B && B != F && D != H ? D : E;
E1 = (D == B && B != F && D != H && E != C) ||
     (B == F && B != D && F != H && E != A) ? B : E;
E2 =  B == F && B != D && F != H ? F : E;
E3 = (D == B && B != F && D != H && E != G) ||
     (D == H && D != B && H != F && E != A) ? D : E;
E4 =  E
E5 = (B == F && B != D && F != H && E != I) ||
     (H == F && D != H && B != F && E != C) ? F : E;
E6 =  D == H && D != B && H != F ? D : E;
E7 = (D == H && D != B && H != F && E != I) ||
     (H == F && D != H && B != F && E != G) ? H : E;
E8 =  H == F && D != H && B != F ? F : E;
```

An illustration of how Scale3x algorithm works is shown in figure 5.

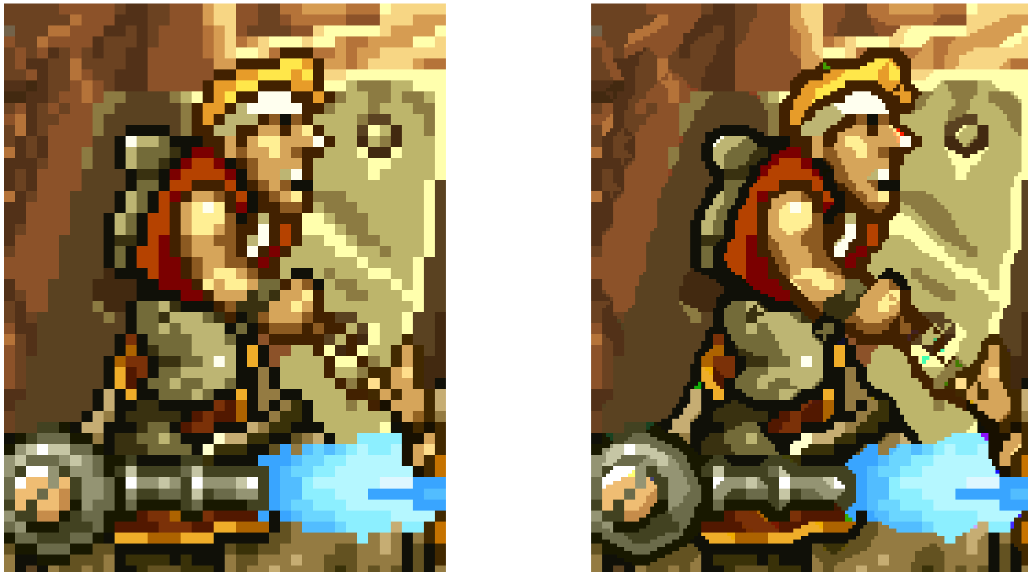


Figure 5: Original picture (zoom 300 %) and Scale3x effect.

3.1.3 Hq3x

The hq3x algorithm/filter was tested by downloading a C++ program from [10]. The algorithm analyses the 9 closest pixels of the source pixel, and sort the pixels in to the categories "close" or "distance" coloured. To categorize the pixels a RGB to YUV conversion is performed to have more tolerance on the Y component. Since there are 8 neighbours, it is 256 predefined combinations. Depending on the neighbours of the source pixel, one of the combinations is used to calculate the colour. The hq3x filter is designed for images with clear sharp edges, and not for photographs.

An illustration of how hq3x algorithm works is shown in figure 6.

Figure 6: Original picture (zoom 300 %) and hq3x effect.

3.1.4 Rogers2x algorithm for hardware

The scale2x algorithm was the inspiration that led to the making of my own test algorithm that scale two times up. This algorithm was made to suit low resolution games. The code for this algorithm can be found in appendix A.3, under the function name **rogers2xAvgGFX_YUV**. First, it was tried to use interpolation with use of RGB colours, but this was not a suitable method. As mention in section 2.1, the human eye respond different to the red, green and blue colour, so when scaling with RGB you should choose different scaling factors on the three colours. By using interpolation with YUV (YC_bC_r) pixel values, the dependencies on the human eye are considered. The interpolation is then straight forward by using the same scaling factor on the three components. But doing RGB to YUV conversion takes some time to calculate. [7] uses the following formulas to do RGB to YUV conversion:

$$\begin{aligned} \text{Y} &= 0.299\text{R} + 0.587\text{G} + 0.114\text{B} \\ \text{Cr} = \text{V'} &= (\text{R-Y})*0.713 \\ \text{Cb} = \text{U'} &= (\text{B-Y})*0.565 \end{aligned}$$

When implementing these formulas in hardware it would use some calculation power, but by doing only left and right shifting of bits if would be fast. So these formulas were introduced to replace the previous:

$$\begin{aligned} \text{Y} &= \text{R}/4 + \text{G}/2 + \text{B}/4 \\ \text{Cr} = \text{V'} &= (\text{R-Y})/2 \\ \text{Cb} = \text{U'} &= (\text{B-Y})/2 \end{aligned}$$

The same was done for YUV to RGB conversion. [7] uses the following formulas to do YUV to RGB conversion:

$$\begin{aligned} \text{R} &= \text{Y} + 1.403\text{V'} \\ \text{G} &= \text{Y - } 0.344\text{U' - } 0.714\text{V'} \\ \text{B} &= \text{Y} + 1.770\text{U'} \end{aligned}$$

These formulas were introduced to replace YUV to RGB conversion:

$$\begin{aligned} \text{R} &= \text{Y} + 2\text{V'} \\ \text{G} &= \text{Y - U'}/2 \text{ - V'}/2 \\ \text{B} &= \text{Y} + 2\text{U'} \end{aligned}$$

Figure 7 illustrates the colour distortion on a picture when using the new conversion formulas. The picture is first converted to YUV, then back to RGB with the new calculation method. The figure shows that the colours appear to be more bright when using the new conversion formulas. The output picture of Lenna is actually very good. The algorithm seems to also suit realistic pictures. This colour distortion may be acceptable when scaling old computer games, but may appear too distorted when doing video scaling.

Figure 7: Illustration colour distortion of Rogers2x algorithm. Original picture (left, zoom 200 %) and Rogers2x algorithm (right).

Like figure 2 for Scale2x algorithm, figure 8 use the same input pixels to calculate the output pixels. The difference is that this algorithm always uses two fourth of the middle input pixel E and one part each of the other two closest input pixels. For instance, to calculate $E0$, 2 parts of E, 1 part of D and 1 part of B are used for interpolation. This ensures that sharp edges are obtained in the scaled picture.

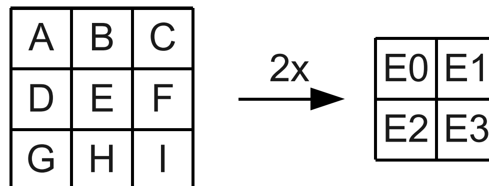


Figure 8: Rogers2x algorithm. Shows expansion of one input pixel.

Figure 9 shows parts of the title screen of the game Wolfenstein 3D. The left picture is the original picture, the middle is scale by Scale2x algorithm, and the right picture is scaled by Rogers2x algorithm.

Figure 9: Original picture (zoom 200 %), Scale2x algorithm, and Rogers2x algorithm.

It is not easy to see whether the Scale2x or Rogers2x produces the best result, but at first glance Rogers2x algorithm may seem more realistic. The major concern is whether Rogers2x algorithm produces to much blur to the output picture. After comparison with many different pictures, Rogers2x algorithm seems to produce a more smooth picture, and more lively. To get a better comparison it would be useful to apply these algorithms to a small sequence video of a low resolution game such as Wolfenstein 3D. The factors that Rogers2x algorithm use (1/4, 1/4 and 2/4) can be toggled to find the best result.

Regardless of which of the two algorithms that produces the best result, both algorithms are surely easy to implement in a hardware solution. The algorithms suits for instance QVGA (320x240) to VGA (640x480) scaling.

3.2 Video scaling algorithms for hardware

Video scaling algorithms were developed in software with great concerns of how it would be implemented in hardware. Three algorithms were developed, and the software implementation can be found in appendix A.1. The three algorithms have the function names **SD2HD**, **HD2FullHD** and **SD2FullHD**.

The first algorithm developed had a scaling factor of 1.25 (SD2HD). This factor is equal to the factor you need to perform SD WIDE to HD up-scaling. To find an appropriate algorithm it was necessary to view the dependencies of the output pixels with regards to the input pixels. Figure 10 shows these dependencies. The division that is done by the straight lines, divided into 5x5 matrix, is the output pixels. The hatching and colouring represents the 4x4 input pixel matrix.

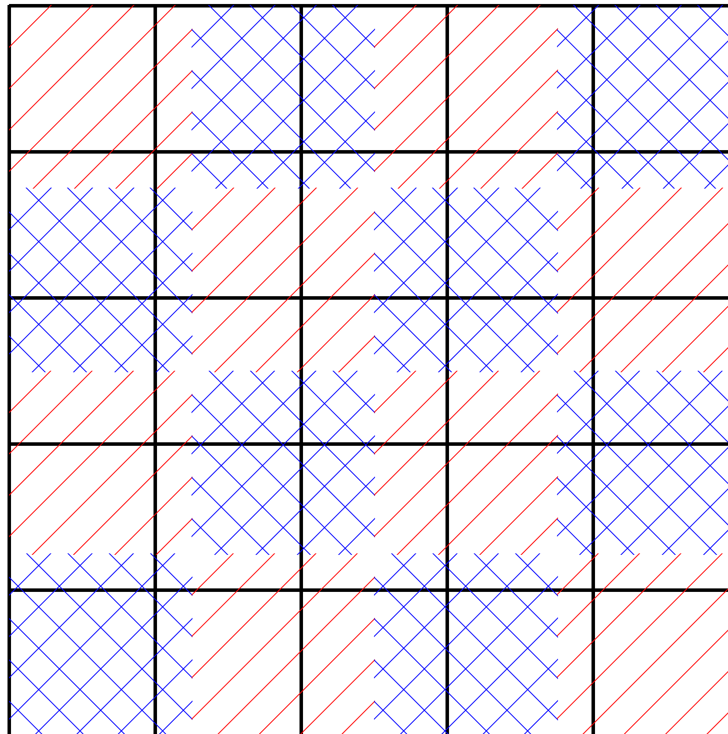


Figure 10: Output dependencies with linear scaling, scaling factor 1.25.

Figure 10 illustrates that the first output pixel (in the up left corner) only depend on the first input pixel. The next output pixel to the right depend on two input pixels, a quarter of the first and three quarters of the next. When studying the dependencies you can see that an output pixel at most depend on four input pixels, the same as bilinear interpolation.

The second algorithm developed had a scaling factor of 1.5 (HD2FullHD). This factor is equal to the factor you need to perform HD to FullHD up-scaling. Like the first algorithm it was necessary to view the dependencies of the output pixels with regards to the input

pixels. Figure 11 shows these dependencies. The division that is done by the straight lines, divided into 3x3 matrix, is the output pixels. The hatching and colouring represents the 2x2 input pixel matrix.

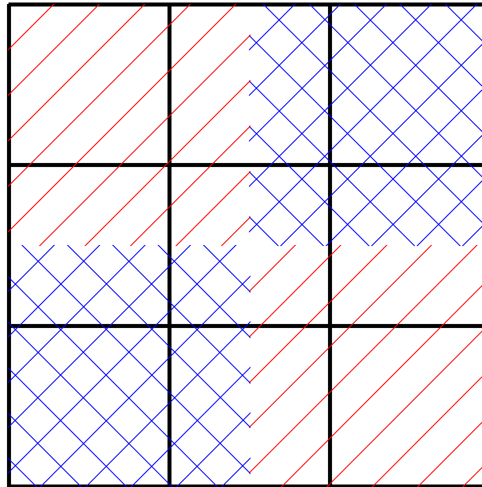


Figure 11: Output dependencies with linear scaling, scaling factor 1.5.

From figure 11 you can see that the first output pixel (in the up left corner) only depends on the first input pixel. The next output pixel to the right depends on two input pixels, one half of the first and one half of the next. When studying the dependencies you can see that an output pixel at most depend on four input pixels.

The third algorithm developed had a scaling factor of 1.875 (SD2FullHD). This factor is equal to the factor you need to perform SD WIDE to FullHD up-scaling. Like in the first and the second algorithm it was necessary to view the dependencies of the output pixels with regards to the input pixels. Figure 12 shows these dependencies. The division that is done by the straight lines, divided into 15x15 matrix, is the output pixels. The hatching and colouring represents the 8x8 input pixel matrix.

From figure 12 you can see that the first output pixel (in the up left corner) only depends on the first input pixel. The next output pixel to the right depends on two input pixels, seven eighth of the first and one eighth of the next. When studying the dependencies you can see that an output pixel at most depend on four input pixels.

When implementing in software, the first two algorithms take use of simple average functions. These average functions are built up with regards of the dependencies described earlier. The functions were thought to be easy to implement in a hardware solution. The algorithms can be found in appendix A.1 under the function names **SD2HD** and **HD2FullHD**.

The third algorithm with the scaling factor of 1.875 was not as straight forward as the two first ones. To create an algorithm it was again needed to take the dependencies in to

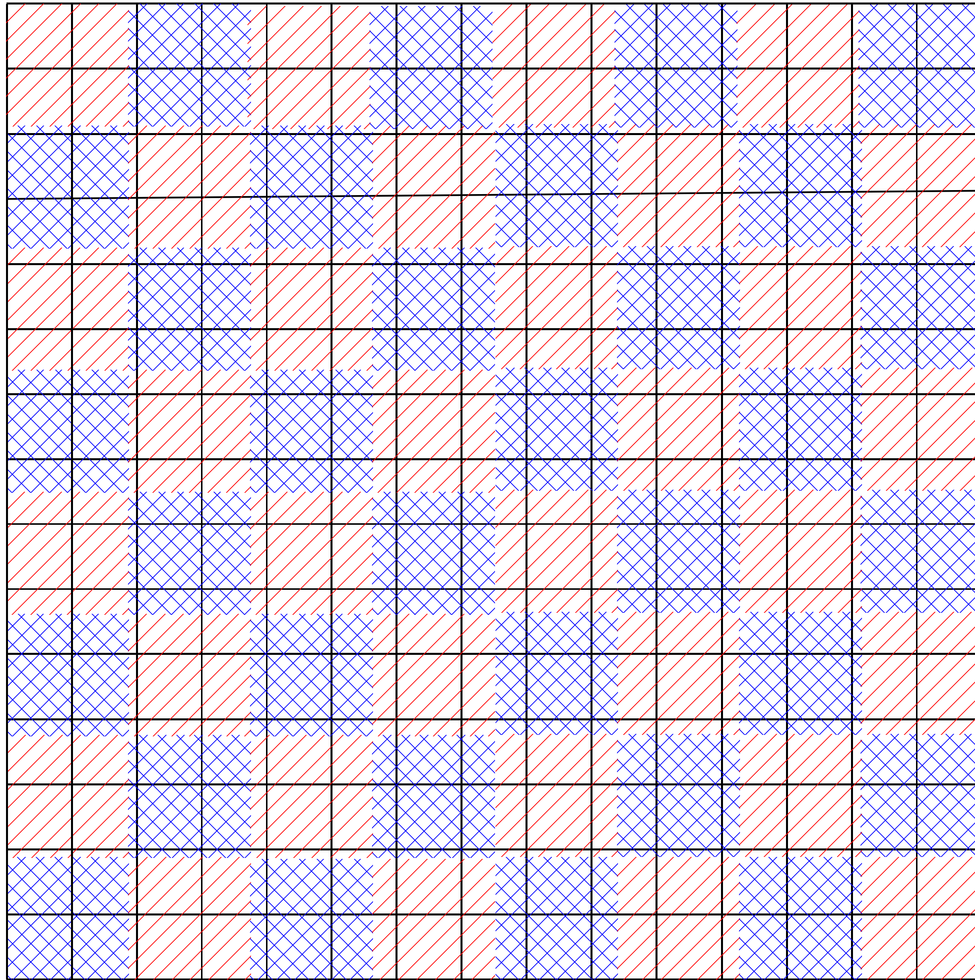


Figure 12: Output dependencies with linear scaling, scaling factor 1.875.

consideration. From figure 12 you can see that more than half of the output pixels depend only on one input pixel. Imagine that the figure has X and Y coordinates starting counting from 0 and up to 14, studying more carefully you can see that it is when X and Y are even the output pixel only depend on one single input pixel. If one of the X or Y is even and not the other, when the output pixels depend on two input pixels. For the remaining output pixels, they all depend on four input pixels.

Algorithm 1 shows how the dependencies are expressed in an algorithm. The software implementation of the algorithm can be found in appendix A.1 under the function name **SD2FullHD**.

Algorithm 1 Linear scaling with scaling factor of 1.875.

Require: $p1, p2, p3, p4$ to be the input pixels that po depends on.
Require: x and y to be modulo of 15 to the real X and Y coordinates of the output frame.

for all po (output pixels) **do**
  if x and y are even **then**
    $po \Leftarrow p1$
  else if y is even **then**
    $po \Leftarrow (p1 * (8 - ((x+1)/2)) + p2 * ((x+1)/2))/8$
  else if x is even **then**
    $po \Leftarrow (p1 * (8 - ((x+1)/2)) + p3 * ((x+1)/2))/8$
  else $\{x$ and y are odd$\}$
    $po \Leftarrow (((x+1)/2) * ((y+1)/2) * p4$
      $+ ((y+1) * 4 - ((x+1)/2) * ((y+1)/2)) * p3$
      $+ ((x+1) * 4 - ((x+1)/2) * ((y+1)/2)) * p2$
      $+ (8 - ((x+1)/2)) * (8 - ((y+1)/2)) * p1)/64;$
  end if
end for

4 AMBA (Advance Microcontroller Bus Architecture)

When implementing the video scaler there was a need for an interface that could read the input frame from memory, and write the scaled frame to memory. A configuration interface was also needed. ARM's Advance eXtensible Interface (AXI) and the Advance Peripheral Bus (APB) were selected for these tasks. In this section both AXI and APB are briefly described. The complete specification can be found on-line at ARM Information Center [1]. Three modules were made with regards to the AMBA specification, APB module, AXI Read module and AXI Write module, these are described in this section.

4.1 AMBA APB (Advanced Peripheral Bus)

The AMBA APB is designed for low bandwidth control accesses. The APB module used in this project works as a slave and has to comply to an APB bridge that works as a master. The APB module is used for setting registers that is relevant for video scaling, for instance, registers containing size of input and output frame.

The AMBA APB has an slave signal called `PSEL` that specifies whether the APB is selected or not. When the `PSEL` is asserted, data transfer can occur. The APB specifies a signal called `PWRITE` that indicates an APB write when HIGH, and an APB read when LOW. There are two data buses, one for the APB read which is driven by the slave, and one for the APB write that is driven by the APB bridge. The slave use a ready signal to indicate that it is ready to receive or send data. This is default HIGH for the implemented APB module. An enable signal driven by the APB bridge is asserted the next clock cycle after `PSEL`, and the transfer occurs when the ready signal and the enable signal is HIGH. APB also specifies an address bus that holds the information of where the data is to be written to, or read from.

The AMBA APB is built up as a finite state machine with idle, setup and access states. How the state machine works can be found in [1].

4.1.1 APB module

The APB module was the first module constructed for the implementation. The Verilog code for the APB module is shown in appendix B.3. The entity of the module is as follows:

```
module apb_slv
  (PCLK, PRESETn,
  PADDR, PSEL, PENABLE, PWRITE, PWDATA, PREADY, PRDATA, PSLVERR,
  start_addr, dest_addr, mem_size_in, mem_size_out, start_scale,
  frame_width_in, frame_width_out, frame_height_out, irq_vs);

// Clock and Reset
```

```
input           PCLK;
input           PRESETn;
// APB Slave interface
input   [31:0]   PADDR;
input            PSEL;
input            PENABLE;
input            PWRITE;
input   [31:0]   PWDATA;
output  [31:0]   PRDATA;
output           PREADY;
output           PSLVERR;
// Signals connections to other modules:
output  [31:0]   start_addr;
output  [31:0]   dest_addr;
output  [15:0]   mem_size_in;
output  [15:0]   mem_size_out;
output           start_scale;
output  [10:0]   frame_width_in; // max 1920
output  [10:0]   frame_width_out; // max 1920
output  [10:0]   frame_height_out; // max 1080
input            irq_vs;
```

The APB module module should give the other modules in the video scaler information by assigning registers a certain value. These registers are set for instance depending on how large the input frame and output frame is. When writing to the registers the PADDR signal must be set correctl. Below it is shown what addresses that assign what registers:

```
`define APB_START_ADDR    12'h00 // Physical start address
                                 // output [31:0]  start_addr_r
`define APB_DEST_ADDR     12'h04 // Physical destination address
                                 // output [31:0]  dest_addr_r
`define APB_MEM_IN_SIZE   12'h08 // Size to read
                                 // output [15:0]  mem_size_in_r;
`define APB_MEM_OUT_SIZE  12'h0c // Size to write
                                 // output [15:0]  mem_size_out_r;
`define APB_START_SCALE   12'h10 // To start scaling
                                 // output         start_scale_r;
`define APB_FRAME_W_IN    12'h14 // Input frame width
                                 // output [10:0]  frame_width_in_r;
`define APB_FRAME_W_OUT   12'h18 // Output frame width
                                 // output [10:0]  frame_width_out_r;
`define APB_FRAME_H_OUT   12'h1c // Output frame height
                                 // output [10:0]  frame_height_out_r;
`define APB_TOTAL_COUNT   12'h20 // Read total count
                                 // reg    [23:0]  total_clk_count;
```

For instance, when assigning the register that holds the amount of bytes in the input frame (APB_MEM_IN_SIZE), the PADDR has to be assigned to the hexadecimal value 12'h08, and then the write data bus (PWDATA) must contain the amount.

The signal description of the signals connected to other modules in the design are described in table 1, for the remaining signals, see [1]. All the output signals are connected to the registers previous described.

Table 1: APB - Signal connected to the video scaler

SIGNALS	DESCRIPTION
start_addr [31:0]	Physical start address. Connected to the AXI Read module. Sets the first memory location to read from when **start_scale** is asserted.
dest_addr [31:0]	Physical destination address. Connected to the AXI Write module. Sets the first memory location to write to when **start_scale** is asserted.
mem_size_in [15:0]	Memory size of input frame, number of addresses to read from. Connected to the AXI Read module so that the module knows when to stop reading data from new addresses.
mem_size_out [15:0]	Memory size of output frame, number of addresses to write to. Connected to the AXI Write module so that the module knows when to stop writing data to new addresses.
start_scale	Start scaling frame. This signal is connected both the AXI Read module and the AXI Write module. It signals the modules to start reading and writing addresses. All other signals/registers described in this tabel must be assign before asserting **start_scale**.
frame_width_in [10:0]	Frame width of input. Connected to mem_reg module.
frame_width_out [10:0]	Frame width of output. Connected to scale module.
frame_height_out [10:0]	Frame height of output. Connected to scale module.
irq_vs	Interrupt. Input from the AXI Write module. When **irq_vs** is HIGH the frame is finished scaled and written to memory.

4.2 AMBA AXI (Advance eXtensible Interface)

The AXI was implemented with help of the AMBA AXI Protocol Specification from ARM Information Center. Only a small part of the specification is described in this section, see [1] for the whole specification.

To get a brief understanding on how the AXI works it is useful to understand how the handshake procedures works. The handshake is between a VALID signal and a READY signal. There are five channels in AXI that uses this VALID/READY handshake to transfer data and control information. The source generates a VALID signal to indicate when the data or control information is available. The destination accepts the data or control information by asserting a READY signal. Transfer occurs only when both VALID and READY signals are HIGH.

There are three ways a handshake can occur, either VALID or READY can be HIGH first, or they can be set HIGH in the same clock cycle which leads to immediately transfer. For the two other ways, if VALID is HIGH before READY the source have to hold the data or control information stable until the destination drives READY HIGH. It is the same when READY is HIGH before VALID, then the destination have to wait for the data or control information to be presented by the source in form of the signal VALID.

4.2.1 AXI Read module

An implementation in Verilog of an AXI Read module is shown in appendix B.1. The AXI Read module includes all the signals concerning reading from the AMBA AXI bus. The AXI Read module works as a master on the bus and communicate with a slave.

The AXI Read module reads data from addresses, it drives the address bus `ARADDR`. When an address is valid, the `ARVALID` signal is asserted, then the data belonging to the address is ready to be read. The data that the AXI Read module reads is the input frame. An examples of how the read transaction works is shown in figure 13. This example reads a burst. A burst can contain 1 to 16 data packets, so by issuing one address you can receive up to 16 data packets. In this design a burst length of 8 was used.

Figure 13 shows how the AXI Read module can drive another burst address after the slave accepts the first address. This enables the slave to begin processing data for the second burst in parallel with the completion of the first burst. The AXI Read module drives `ARADDR`, `ARVALID` and `RREADY`. The read ready signal, `RREADY`, indicates whether the AXI Read module is ready to receive data.

The slave keeps the `RVALID` signal LOW until the read data is available. For the final data transfer of the burst, the slave asserts the `RLAST` signal to indicate that the last data packet is being transferred. The read data bus, `RDATA`, has a width of 64 bits.

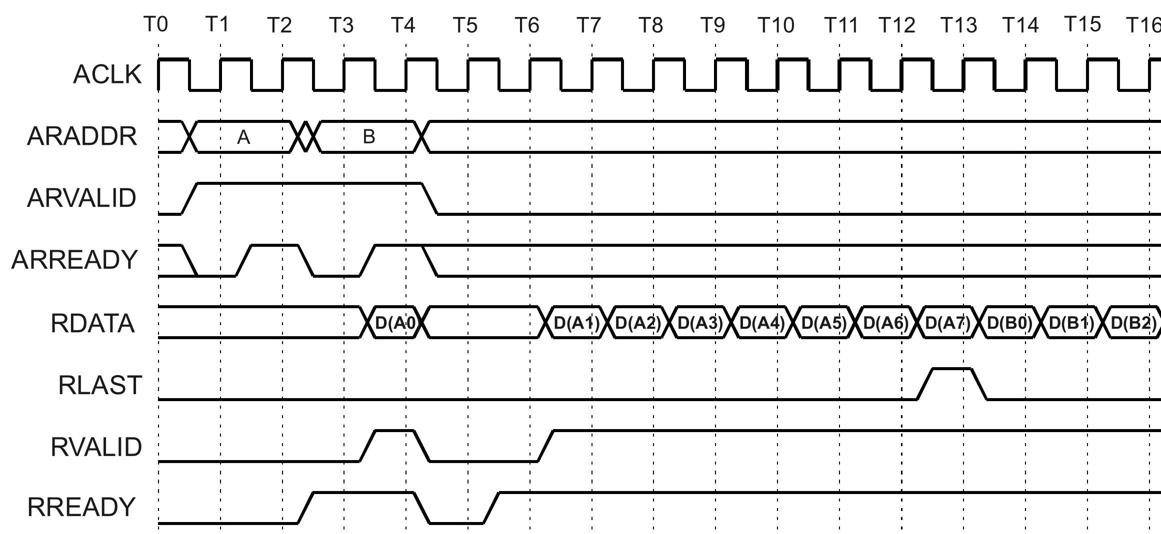


Figure 13: AXI read overlapping burst example (burst of 8 transfers).

The AXI Read module transfers the input frame to a RAM module in the video scaler. When the video scaler was developed there was a need for stopping the data the AXI Read module transferred to the RAM. To prevent overwriting of useful data in the RAM a `stop_transfer` signal controlled by the memory control module (mem_reg module) is used. The mem_reg module is described in section 6.2.

The following Verilog code shows how the `stop_transfer` signal stops the reading of new addresses.

```
// Stop reading in new addresses?
if (start_scale_r) // Start scale has been set.
  arvalid_r <= ~stop_transfer; // stop_transfer, input from mem_reg.
```

The `arvalid_r` register is wired to the `ARVALID` signal that indicates whether an address is valid to read.

4.2.2 AXI Write module

An implementation in Verilog of an AXI Write module is shown in appendix B.2. The AXI Write module includes all the signals concerning writing from the AMBA AXI bus. The AXI Write module works as a master and communicate with a slave AXI.

The AXI Write module writes addresses to a slave AXI, it drives the address bus `AWADDR`. The slave AXI can receive addresses when it asserts `AWREADY`. After receiving an address it prepares for receiving data from the AXI Write module on the write data bus, `WDATA`. When data is valid on the bus, the AXI Write module asserts the `WVALID` signal. The slave AXI can receive data when `WREADY` is HIGH. The data written to the memory is the

output frame. A write burst example is shown in figure 14. Like the AXI Read module, the burst length is 8.

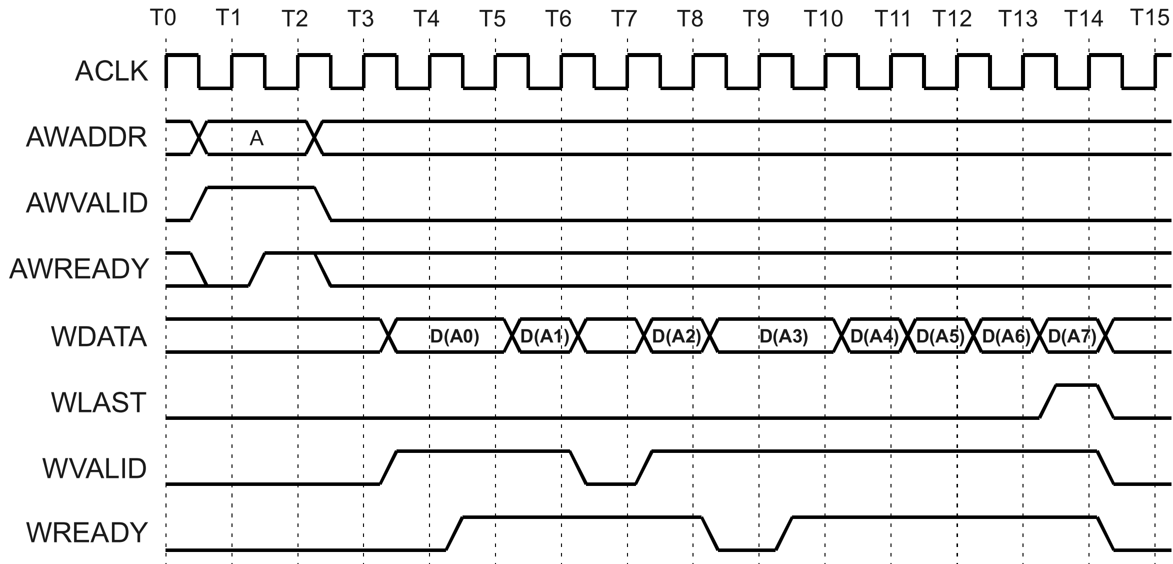


Figure 14: AXI write burst example (burst of 8 transfers).

When the AXI Write module sends the last data packet, it sets the `WLAST` signal HIGH. The write data bus, `WDATA`, has a width of 64 bits. `AWREADY` and `WREADY` are input signals from the slave AXI, the rest is output signals from the AXI Write module.

The AXI Write module receives data from a FIFO in the video scaler system. The AXI Write module only starts writing data to the memory when the FIFO holds a whole burst. The FIFO provides a `data_count` signal that contain the amount of data packets in the FIFO, so when the amount is the burst length or more, the writing can occur. To signal the last data packet a simple counter is used, counting down from the burst length to zero.

An interrupt signal `irq_vs` is asserted when a frame is finished scaled. This signal is wired to the `IRQ` signal described in section 6, *Implementation in hardware*. The Verilog code below shows how the `irq_vs` is assigned:

```
// IRQ when: FIFO is empty, finished writing addresses, scale module
   finished.
assign irq_vs   = empty && addr_finished_r && frame_finished_r;
```

The interrupt signal is asserted when the FIFO is empty, the AXI Write module is finished writing addresses, and the scaling module is finished scaling the frame.

5 Verilog testbench

The verilog testbench was provided by ARM, and is used for simulating the video scaler. The testbench should provide a software simulation equal to how the hardware should behave. The essential modules that has to be included, and work correctly, is the AXI and APB modules. All communication is through AXI and APB.

Figure 15 shows an overview of how the testbench works.

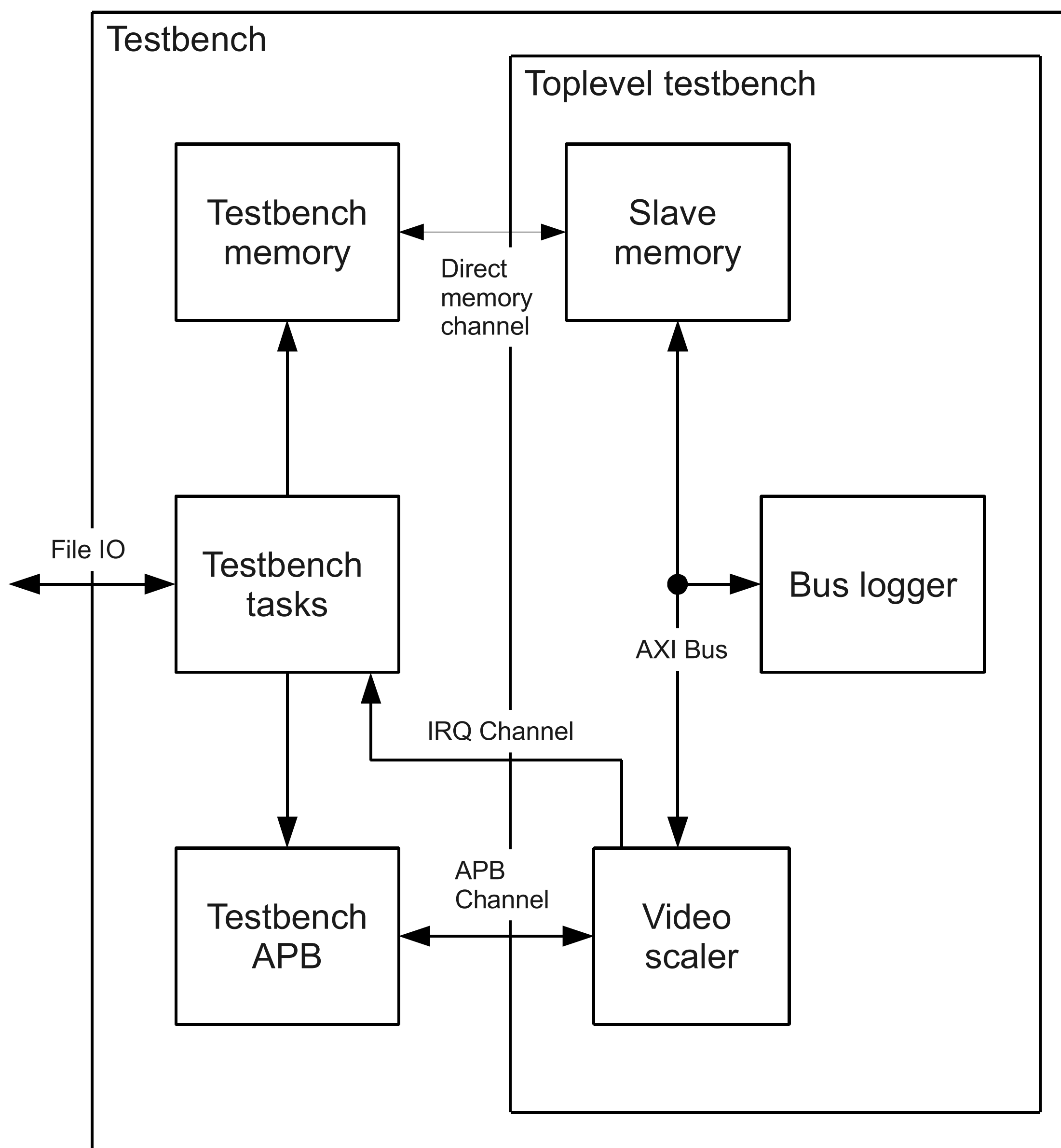


Figure 15: Verilog toplevel testbench structure.

The testbench is a command-controlled testbench. Communication with the testbench

happens by setting up a configuration file. The configuration file, *config.txt*, can include for instance commands to:

- setting/reading the registers in APB
- loading frames in to memory
- reading from memory
- resetting modules
- handle interrupt (reading from APB)
- setting read and write latency
- setting allowed outstanding read/write transactions
- enabling buslog

To be certain that an FPGA implementation is going to work it is necessary to experiment with the read and write latency, setting them to several different values. The outstanding transactions also needs to be experimented with. Outstanding transactions is the number allowed addresses that AXI can have outstanding. The read and write acceptance capabilities of the slave memory model can be set in *config.txt*.

6 Implementation in hardware

This section describes how the video scaler system is built up. First it is described how AXI and APB are tested to work correctly, then follows a description of the modules in the video scaler. The two last sub-sections describes how the video scaler was tested and implemented on the FPGA.

6.1 Testing AXI and APB

To test the implementation of the three modules AXI Read, AXI Write and APB, a simple test set-up was used, this is illustrated in figure 16. By doing this simple test it was possible to confirm that handshakes routines and flagging of the last data packet were correct.

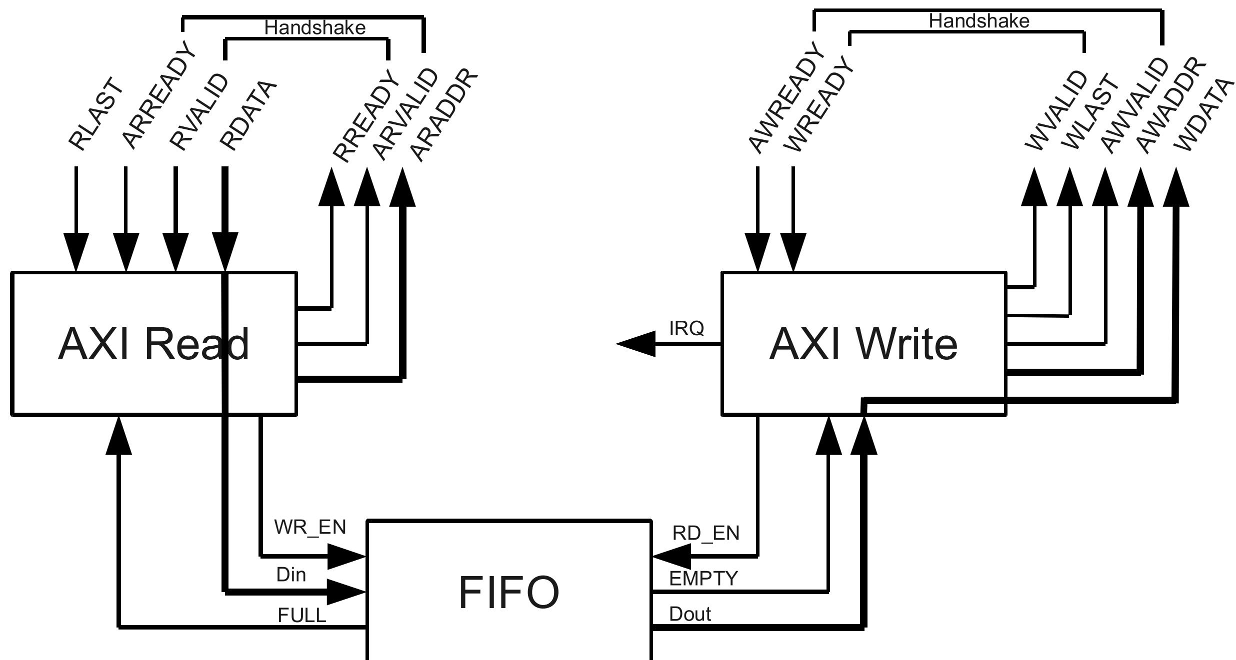


Figure 16: Test of AXI and APB interface.

The testbench in section 5, *Verilog testbench*, was used on this simple test set-up.

The APB module is not shown in the figure. The APB module contributes with the signals `start_addr` (the start address) and `mem_size_in` (amount of memory to read) to the AXI Read module. It also contribute with the signals `dest_addr` (the destination address) and `mem_size_out` (the amount of memory to write) to the AXI Write module. `start_addr` and `dest_addr` is the physical address in memory.

To confirm that the modules worked properly, the input frame had to be the same as the output frame, since no scaling were applied. Naturally, the test did not pass on the first try,

so diagnostics tools were essential. When enabling buslog from *config.txt*, both data and addresses from AXI bus could be traced. The diagnostic tool that had the best effect was the vector wave view. All the vectors for input, output and internal registers to all modules were logged. These vectors could be viewed in the Discovery Visualization Environment (DVE) from Synopsys verification solution (VCS) [18]. With DVE it was possible to easily find where errors occurred.

6.2 Memory control module (mem_reg)

The memory control module was made simultaneously with the Scale 1.25 module because the two modules are depending on each other. It was decided to use two internal registers that could contain four 8 byte-blocks since the input data are represented in 8 byte-blocks (64 bit data bus). The reason for using four 8 byte-blocks is described in the two next sections. Data from a internal RAM is used to assign the internal registers. The Verilog code for the RAM module is shown in appendix B.7.

The memory control module's main task is to keep the internal registers for the Scale 1.25 module and the Scale 1.875 module up to date. When the mem_reg module receives a HIGH `reg_shift` signal from the scaling module, it states that the internal registers can shift in new data. The mem_reg module communicates back to the scaling module by asserting `reg_ready`, stating that the internal registers are ready to be used, and calculation can proceed.

The Verilog code below shows how mem_reg module updates the internal registers `l1` and `l2` when it receive a HIGH `reg_shift` from the scaling module. It also ensures that the scaling pauses if the FIFO is full or the write address and read address is the same on the RAM. The mem_reg module asserts `reg_ready` only when finished updating both internal registers.

```
// Normal sequence:
 else if (aw_r != ar_r && reg_shift && !FIFO_full) begin

   // Read part of Line1 from RAM
   if (on_line1) begin
     on_line1 <= 0;
     reg_ready_r <= 0;
     l1[reg_pos] <= ram_data;
     ar_r <= ar_r + LINE_WIDTH; // Read second line next
   end

   // Read part of Line2 from RAM
   else begin
     on_line1 <= 1;
     reg_ready_r <= 1;
     l2[reg_pos] <= ram_data;
     count_line_width <= count_line_width + 1;
```

```
      ...
      if (reg_pos == 3) // reg_pos: 0->1->2->3->0->1..
        reg_pos <= 0;
      else
        reg_pos <= reg_pos + 1;
    end
    ...
  end
```

There is a slightly difference between the mem_reg module needed for the different scaling factors 1.25 and 1.875. The difference is the number of lines needed. With a scaling factor of 1.25 the number of lines needed is 5. After 5 lines, symmetry is reached. For scaling factor 1.875 the number of lines need to be 15, since symmetry is reached at 15. The following Verilog code shows how the read address on the RAM is assigned when the mem_reg module is finished reading a line in the input frame. The case sentence is for the scaling factor 1.25. When scaling with the factor 1.875 the case sentence have to be changed, and include 15 cases.

```
// On line 2:
if (count_line_width == LINE_WIDTH-1) begin
  count_line_width <= 0;
  line_number <= line_number + 1;

  case (line_number)
    3'b000: ar_r <= ar_r - LINE_WIDTH - LINE_WIDTH + 1;
    3'b001: ar_r <= ar_r - LINE_WIDTH + 1;
    3'b010: ar_r <= ar_r - LINE_WIDTH + 1;
    3'b011: ar_r <= ar_r - LINE_WIDTH - LINE_WIDTH + 1;
    3'b100: begin
      ar_r <= ar_r + 1;
      line_number <= 0;
    end
  endcase
end
```

Viewing figure 10 on page 12 may be useful to understand this code. When `line_number` is zero and the mem_reg module is finished reading a whole input line, the read address is set to the start address of the input line that was just read. This is because the pixels on the first and second output line both depend on the pixels in the first input line. This is also the case when the `line_number` is 3. Symmetry is reached on the last line, then no previous input pixels are reused.

As mentioned, the mem_reg module uses a RAM to read from. The RAM contains data provided by the AXI Read module. To prevent the AXI Read module to overwrite useful information in the RAM, the mem_reg module needs to stop the AXI Read module at any time. This is provided by an output signal `stop_transfer`. When the AXI Read module receives a HIGH `stop_transfer` it stops the process of issuing new addresses temporarily, until `stop_transfer` goes LOW again. This ensures that useful data in

RAM is not overwritten. Only a small amount of data is written to the RAM when stop_transfer is HIGH. This small amount is the outstanding data that is already in progress when the AXI Read module issues a new address. See section 4.2.1 for more information on the AXI Read module. Verilog code for stop_transfer:

```
// Control of RAM (ar must be behind aw):
if (on_line1) begin
  if (aw_r > ar_r + 256) // Must contain at least 2 input lines
    stop_transfer_r <= 1; // Stop reading in new addr on AXI read
  else
    stop_transfer_r <= 0;
end
```

The main challenge when constructing the mem_reg module and the video scaler was the concern of reusing input pixels. A simple way to do the design could have been to read in input data as many times as you should use them. But this would be inefficient. The system would be slower, the bandwidth would increase and power consumption would be high. Since the video scaler is meant to run in real time and use as little area as possible, the design preparation had to be done carefully.

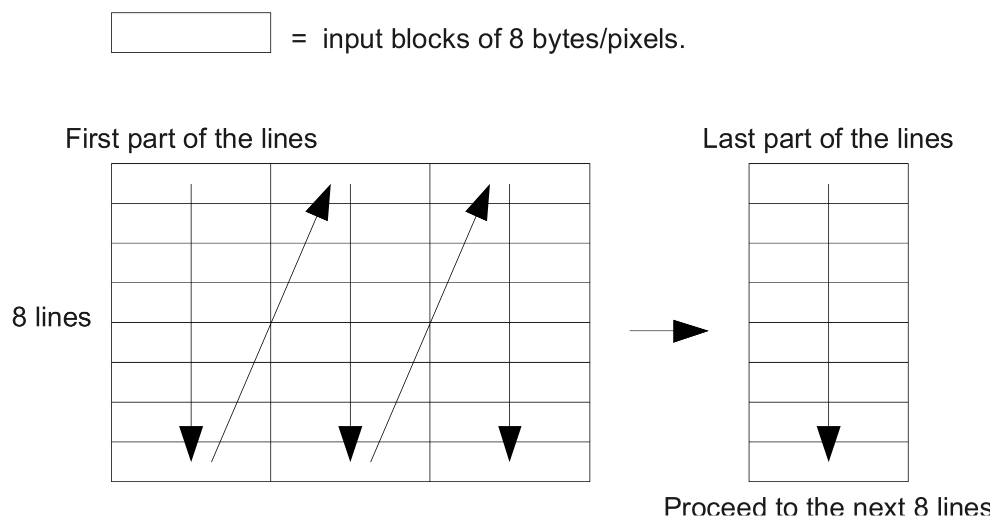


Figure 17: Memory management: Reading from RAM once.

Two different design ideas were discussed to prevent reading in input pixels more than one time. Both ideas stated that input pixels are read by the AXI Read module only once, and stored in a internal RAM. The first idea was to try to read from the RAM only once. The way of doing this was by reading parts of a line in a picture frame, then reading the pixels on next line at the same horizontal position as the previous read. By reading a part of 8 lines and continue with the next part, with the same 8 lines, symmetry was reached. After 8 lines are read the next part increases horizontally. When the last part (the last pixels to the right of the picture) is finished and not useful any more, the next 8 lines becomes the

target to read. A visual representation of the memory management idea is shown in figure 17.

The first idea would give a difficult output for the scaler to handle. Since only parts of a line are given as an input, the output will be of the same line, only containing more bytes. The width of the data bus are set to handle 8 bytes at the time and is not changeable. For instance, an input of 8 bytes from one line with the scaling factor of 1.25 would give an output of 10 bytes for the same line. When converting the output to 8 byte data packets, the first data packet would contain data from the first line, the second data packet would contain data from both the first and the second line, an so on. The problem is visualized in figure 18 for scaling factor 1.25 and figure 19 for scaling factor 1.875.

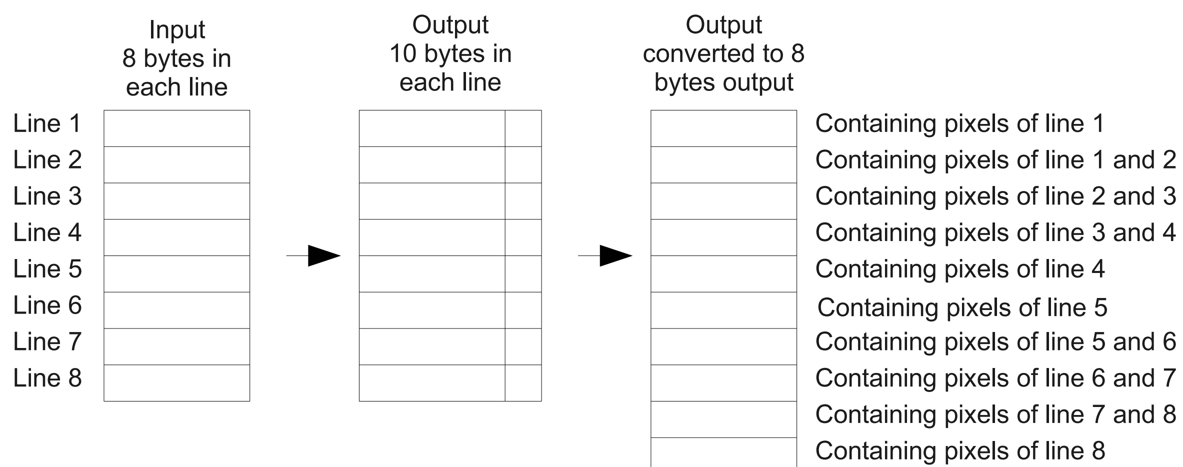


Figure 18: Output of scaling factor 1.25 with one RAM read.

This first idea was considered to have to many drawback versus the benefits it could contribute. The largest drawback was that the memory management in the three modules AXI Read, AXI Write and RAM, would become much more complicated. This would slow down the design phase significantly. The RAM would become very large since a minimum of 16 lines are needed to be stored. Another drawback was that if the frame is represented as scan-lines (the first line arrives first, than the second and so on) a wait for the 8 first lines was needed since you need 8 lines to begin scaling. This would decrease the performance. The benefit of reading only one time from the RAM was considered to be so small in contrast to the major drawbacks it produced. The first idea led to a new way of thinking, reading from the RAM more than one time.

The second idea, and the method that was chosen for the memory control module, was to read from the RAM and write it to an internal register a maximum of two times. To calculate new pixels in the scaling module you need parts of maximum two lines from the original frame. It was decided to use two internal registers containing parts of the first line in one of the registers, and parts of the next line in the other register. By updating these two registers the scaling module could work uninterrupted. The major benefit of this

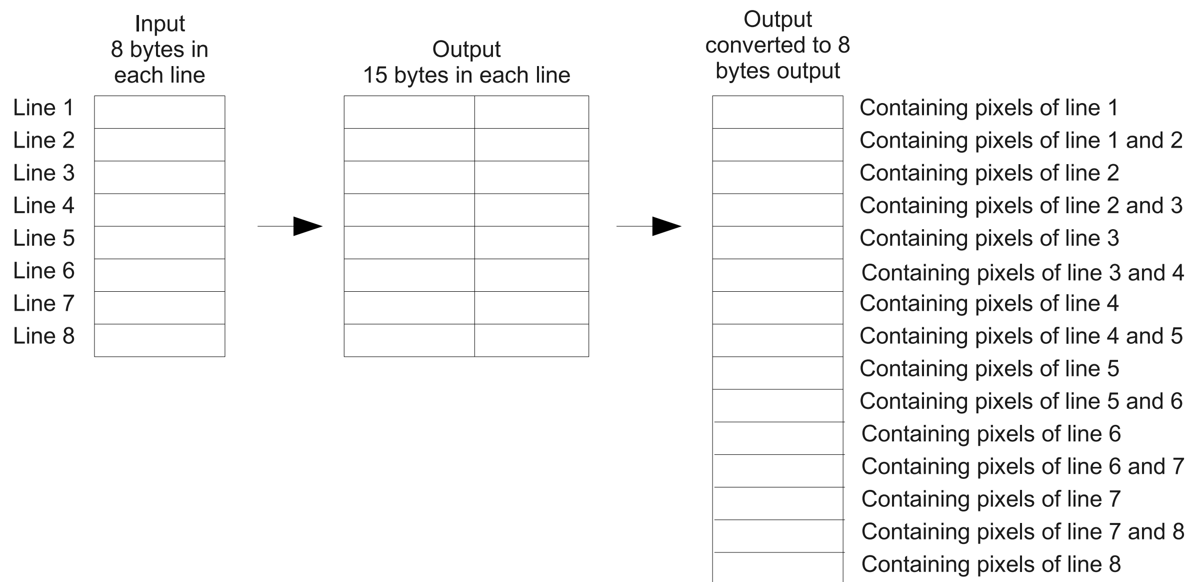


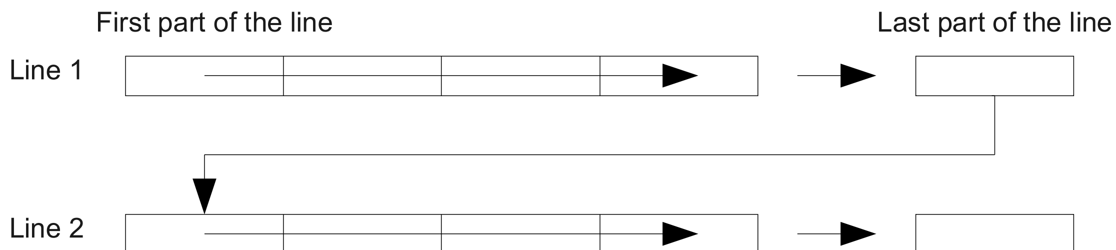
Figure 19: Output of scaling factor 1.875 with one RAM read.

method versus the first is the memory management. Both the AXI Read and the AXI Write use simple incremention on the addresses, and the RAM must contain a minimum of 2 lines, which is 8 times less than the first idea. Another benefit is that if the input frame is represented as scan-lines the scaling can start as soon as the first data packet is received. A drawback is that data has to be read from the RAM twice in to different internal registers.

A visual representation of the memory management method is illustrated in figure 20. The figure shows both internal registers and their contents. If the first internal register contains parts of line 1, when the second should contain parts of line 2, and so on.

6.3 Scale 1.25 (SD WIDE to HD)

The Scale 1.25 module was the first module made for implementation on the FPGA. There were a lot of challenges to meet. One thing of great concern was that data were represented in 8 byte-blocks, and it was a need of two blocks to calculate new output data. With regard to this challenge, a decision to use two internal registers containing four blocks was made. Another solution to use only two blocks as internal registers was rejected due to assumptions that it would increase both code length and size of the module, and further decrease the performance of the module. By using 4 blocks as internal register an option of shifting in new data in advance was introduced. New data could be shifted in as soon as the old data is useless. By choosing this type of internal registers, the registers would always contain useful information for the next calculation of output data.



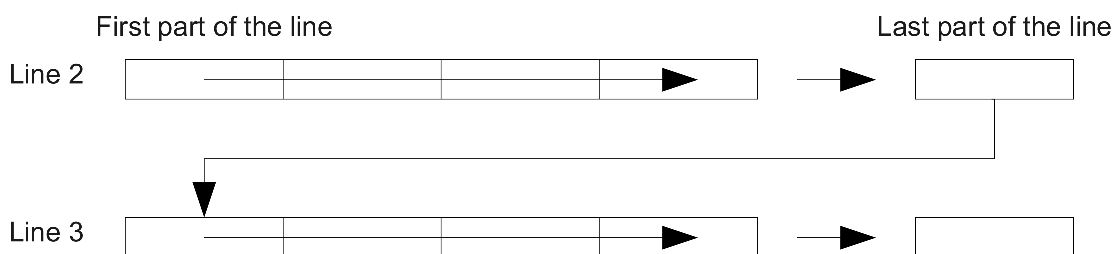


Figure 20: Memory management: Reading from RAM twice.

The scaling factor of 1.25 was the main issue. The verilog testbench demanded that the AXI Write module had a 8 byte wide data bus. For instance, an input of 8 bytes would have an output of 10 bytes. The AXI Write module could not receive 10 bytes of data, only 8 bytes. To deal with this problem the choice of internal register was essential. By using four blocks of 8 bytes as internal register the output of these 32 bytes were 40 bytes. In that way, the AXI Write module could receive 5 data packets of 8 bytes, problem solved. See figure 21.

To transfer the software function of the scaling algorithm SD2HD with scaling factor 1.25 to hardware, a task function in Verilog was used. A task function is a function that can be called more than one time. A solution of using eight task functions was used. One task for each byte in the 8 byte wide data bus. By performing calculations on all of the 8 bytes in parallell, no temporary storing of data was needed before writing data to the FIFO. The task function of the Scale 1.25 module is shown one the next page.

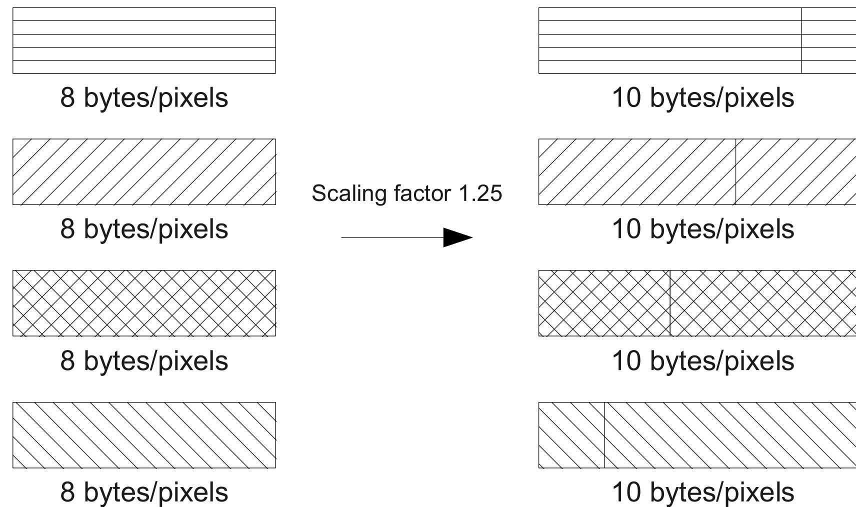


Figure 21: Internal register representation of Scale 1.25 module and output.

```
task avg_func;
  input [4:0] s1_r,s2_r,s3_r,s4_r;         // s1+s2+s3+s4 = 16
  input [7:0] pix1_r,pix2_r,pix3_r,pix4_r; // pixel value
  output [7:0] po;                         // output pixel
  reg [11:0] temp;
  begin

    if      (s1_r[4] == 1'b1) po = pix1_r; // s1_r = 16
    else if (s2_r[4] == 1'b1) po = pix2_r; // s2_r = 16
    else if (s3_r[4] == 1'b1) po = pix3_r; // s3_r = 16
    else if (s4_r[4] == 1'b1) po = pix4_r; // s4_r = 16
    else begin
      temp = ((pix1_r*s1_r[3:0]) + (pix2_r*s2_r[3:0])
            + (pix3_r*s3_r[3:0]) + (pix4_r*s4_r[3:0]));
      po = temp[11:4]; // divide by 16
    end
  end
endtask
```

The *pix* registers holds a pixel value, and the *s* registers states how dependent this input pixel is on the output pixel. $s1_r$ holds the dependence of $pix1_r$, $s2_r$ holds the dependence of $pix2_r$, and so on. The sum of the four *s* registers is 16. To calculate the output pixel, all the *pix* registers are multiplied with its dependence with the output pixel, and summed together before dividing by 16.

The *s* registers are updated outside the task function depending on which line number it is, and the horizontal count. An example of how the *s* registers are assigned is shown in the following Verilog code. In this example it is assumed that it is the first line number, and at the first horizontal count (`line_count`). However, this code runs more than once, due to symmetry. Both `line_number` and `line_count` counts from 0 to 4, and starts

over again. The *a* with a number behind is a predefined constant that has the same bit length as the *s* registers.

```
if (line_number == 3'b000) begin
  if (line_count == 3'b000) begin
    s1[0] <= a16; s2[0] <= a00; s3[0] <= a00; s4[0] <= a00;
    s1[1] <= a04; s2[1] <= a12; s3[1] <= a00; s4[1] <= a00;
    s1[2] <= a08; s2[2] <= a08; s3[2] <= a00; s4[2] <= a00;
    s1[3] <= a12; s2[3] <= a04; s3[3] <= a00; s4[3] <= a00;
    s1[4] <= a00; s2[4] <= a16; s3[4] <= a00; s4[4] <= a00;
    s1[5] <= a16; s2[5] <= a00; s3[5] <= a00; s4[5] <= a00;
    s1[6] <= a04; s2[6] <= a12; s3[6] <= a00; s4[6] <= a00;
    s1[7] <= a08; s2[7] <= a08; s3[7] <= a00; s4[7] <= a00;
  end
  ...
end
```

The predefined constants holds the value behind *a*. For instance, a08 is defined as the vector ”01000”. The four closest input pixels is used when calculation a new pixel. $s1$ is the register that holds the value of dependence for the closest pixel up to the left, $s2$ holds the value of dependence for the closest pixel up to the right, and so on. The sum of the *s* registers is always 16. There are 8 cases of the *s* registers because 8 output pixels are calculated.

As mentioned in section 6.2, *Memory control module*, the scaling module and the memory control module communicate by asserting and de-asserting `reg_ready` and `reg_shift`. How `reg_ready` works is already explained, but `reg_shift` also needs an explanation. The Verilog code below shows how `reg_shift` is asserted and de-asserted.

```
// else if(posedge ACLK)
else begin

  if (FIFO_full) // If FIFO is full hold last reg_shift value
    reg_shift_r <= reg_shift_r;
  else
    reg_shift_r <= 1'b1; // Default value

  // Start
  if (reg_ready || last_data_packet)
  begin

    if (line_count == 3'b000)        // Horizontal count 1
      reg_shift_r <= 1'b0;
    else if (line_count == 3'b001)   // Horizontal count 2
      // SHIFT IN NEW DATA IN [255:192]
      reg_shift_r <= 1'b1;
    else if (line_count == 3'b010)   // Horizontal count 3
      // SHIFT IN NEW DATA IN [191:128]
      reg_shift_r <= 1'b1;
    else if (line_count == 3'b011)   // Horizontal count 4
      // SHIFT IN NEW DATA IN [127:64]
```

```
        reg_shift_r <= 1'b1;
      else if (line_count == 3'b100)   // Horizontal count 5
        // SHIFT IN NEW DATA IN [63:0]
        reg_shift_r <= 1'b1;
    ...
    end
  ...
  end
```

As the Verilog code shows, when horizontal count is 1 the parts of the internal registers is also needed in the next calculation, `reg_shift` is assigned LOW. When the mem_reg module receives a LOW `reg_shift` it holds `reg_ready` HIGH since no shifting of the registers is needed. A new calculation of output data follows immediately. For the other horizontal count values it is needed to shift in new data in to the internal registers. How the `reg_shift` is assigned is also visualized in figure 21 on page 31.

When the FIFO is full the mem_reg module holds `reg_ready` LOW so that the scaling module do not calculate new output data. To start where the two modules left of where is a need to hold the previous `reg_shift` value then the FIFO is full.

6.4 Scale 1.875 (SD WIDE to FullHD)

A lot of time were spent on the Scale 1.25 module and mem_reg module (Memory control module) to make a scalable system. When the Scale 1.25 module and the rest of the system worked properly it was time to scale by a different factor. A few changes to the mem_reg module had to be made. The number of lines had to be increased from 5 to 15. This change is described in section 6.2, *Memory control module*. The symmetry is first reached after finishing line number 15. The first and the sixteenth output line uses the same calculation method. It is the same for every line, mathematically expressed, the i'th line uses the same calculation as the i'th + 15.

A task function was also used when scaling algorithm SD2FullHD was transferred to hardware. Like scaling factor 1.25, the hardware solution of the scaling factor 1.875 also use eight task functions for the same reasons. The following Verilog code shows the task function of the Scale 1.875 module.

```
task odd_func;
  input [3:0] x_r,y_r; // max 4'b1110 in
  input [7:0] pix1_r,pix2_r,pix3_r,pix4_r; // pixel value
  output [7:0] po;

  reg [3:0] x_plus1;
  reg [3:0] y_plus1;
  reg [2:0] x_plus1_div2;
  reg [2:0] y_plus1_div2;
  reg [5:0] x_plus1_mult4;
  reg [5:0] y_plus1_mult4;
```

```
reg [6:0] x_mult_y_div2;
reg [3:0] eight_minus_x_plus1_div2;
reg [3:0] eight_minus_y_plus1_div2;

reg [7:0] po_when_both_modulo;

reg [10:0] modulo_arg1;
reg [7:0] modulo_arg_pix1;
reg [7:0] modulo_arg_pix2;
reg [3:0] modulo_arg_8minus;
reg [2:0] modulo_arg_x_or_y;

reg [13:0] temp1, temp2, temp3, temp4;
reg [13:0] temp_res;

begin
  // Temporary registers:
  x_plus1 = x_r + 1;                            // x+1
  y_plus1 = y_r + 1;                            // y+1
  x_plus1_div2 = x_plus1 [3:1];                 // (x+1)/2
  y_plus1_div2 = y_plus1 [3:1];                 // (y+1)/2
  x_plus1_mult4 = {x_plus1, 2'b0};              // (x+1)*4
  y_plus1_mult4 = {y_plus1, 2'b0};              // (y+1)*4
  eight_minus_x_plus1_div2 = 8 - x_plus1_div2; // 8-(x+1)/2
  eight_minus_y_plus1_div2 = 8 - y_plus1_div2; // 8-(y+1)/2
  x_mult_y_div2 = x_plus1_div2 * y_plus1_div2; // ((x+1)/2)*((y+1)/2)

  // Set the right registers when x and y is even:
  if (y_r == 14)
    po_when_both_modulo = pix3_r;
  else
    po_when_both_modulo = pix1_r;

  // Set registers when x or y is even:
  if (y_r[0] == 0) begin
    if (y_r == 14) begin
      modulo_arg_pix1 = pix3_r;
      modulo_arg_pix2 = pix4_r;
    end
    else begin
      modulo_arg_pix1 = pix1_r;
      modulo_arg_pix2 = pix2_r;
    end
    modulo_arg_8minus = eight_minus_x_plus1_div2;
    modulo_arg_x_or_y = x_plus1_div2;
  end

  else begin
    modulo_arg_pix1 = pix1_r;
    modulo_arg_pix2 = pix3_r;
    modulo_arg_8minus = eight_minus_y_plus1_div2;
```

```
      modulo_arg_x_or_y = y_plus1_div2;
    end

    // This is use for both averaging with 2 and 4 pixels:
    // (x or y is even) or (x and y is odd)
    modulo_arg1 = modulo_arg_8minus * modulo_arg_pix1;

    // Temp registers for when x and y is odd:
    temp4 = x_mult_y_div2 * pix4_r;
    temp3 = (y_plus1_mult4 - x_mult_y_div2) * pix3_r;
    temp2 = (x_plus1_mult4 - x_mult_y_div2) * pix2_r;
    temp1 = eight_minus_x_plus1_div2 * modulo_arg1;

    // Determine output value:
    if (x_r[0] == 0 && y_r[0] == 0) // (x_r and y_r are even)
      po = po_when_both_modulo;

    else if (y_r[0] == 0 || x_r[0] == 0) begin // (x_r or y_r are even)
      temp_res = modulo_arg1 + (modulo_arg_x_or_y * modulo_arg_pix2);
      po = temp_res [10:3]; // Divide by 8
    end

    else begin // (x_r and y_r are odd)
      temp_res = temp1 + temp2 + temp3 + temp4;
      po = temp_res[13:6]; // Divide by 64
    end
  end
endtask
```

In the Verilog code it is included many registers for temporarily storing values. The registers are used because there are a lot of reuse of the same calculations. It may be hard to see what the Verilog code does, but if you compare it with algorithm 1 on page 15 and the C++ code in appendix A.1 you will find it to be exactly the same. The task function was tested with several approaches to find the one that produced, hopefully, the smallest area.

The Scale 1.875 module uses the same method to assign `reg_shift` as the Scale 1.25 module to tell the mem_reg module to shift in new data in the internal registers. The only difference is that the horizontal count is up to 15, not 5 as Scale 1.25 module uses. How the `reg_shift` is assigned is visualized in figure 22.

Figure 22 shows that the first output (suppose we start at the top of the picture) is dependent on only the first input block. The second output block is also dependent on the first input block, so we can not shift in new data in the internal registers. The third output block is only dependent on the second input block, so now we can shift in new data (the fifth block) where the first block was, and so on.

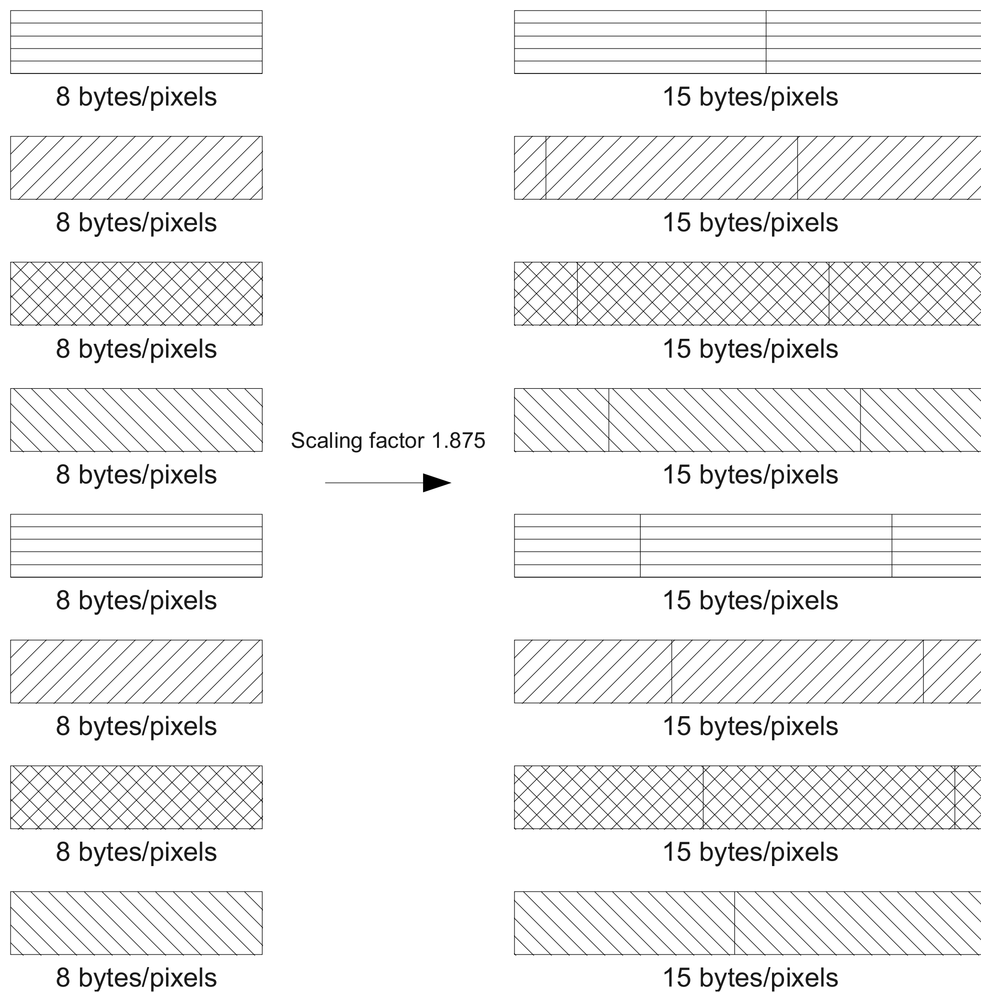


Figure 22: Internal register representation of Scale 1.875 module and output.

6.5 Test of video scaler system

Figure 23 illustrates how the video scaler system is built up. The figure includes three of the same modules that were tested in section 6.1, *Test AXI and APB* on page 24. The system is expanded by a scale module, a RAM module, a FIFO, and a mem_reg module. The RAM module is a 2 port RAM, all other modules are described earlier. The figure only contain the most relevant signals. For viewing all the signals, see appendix C.

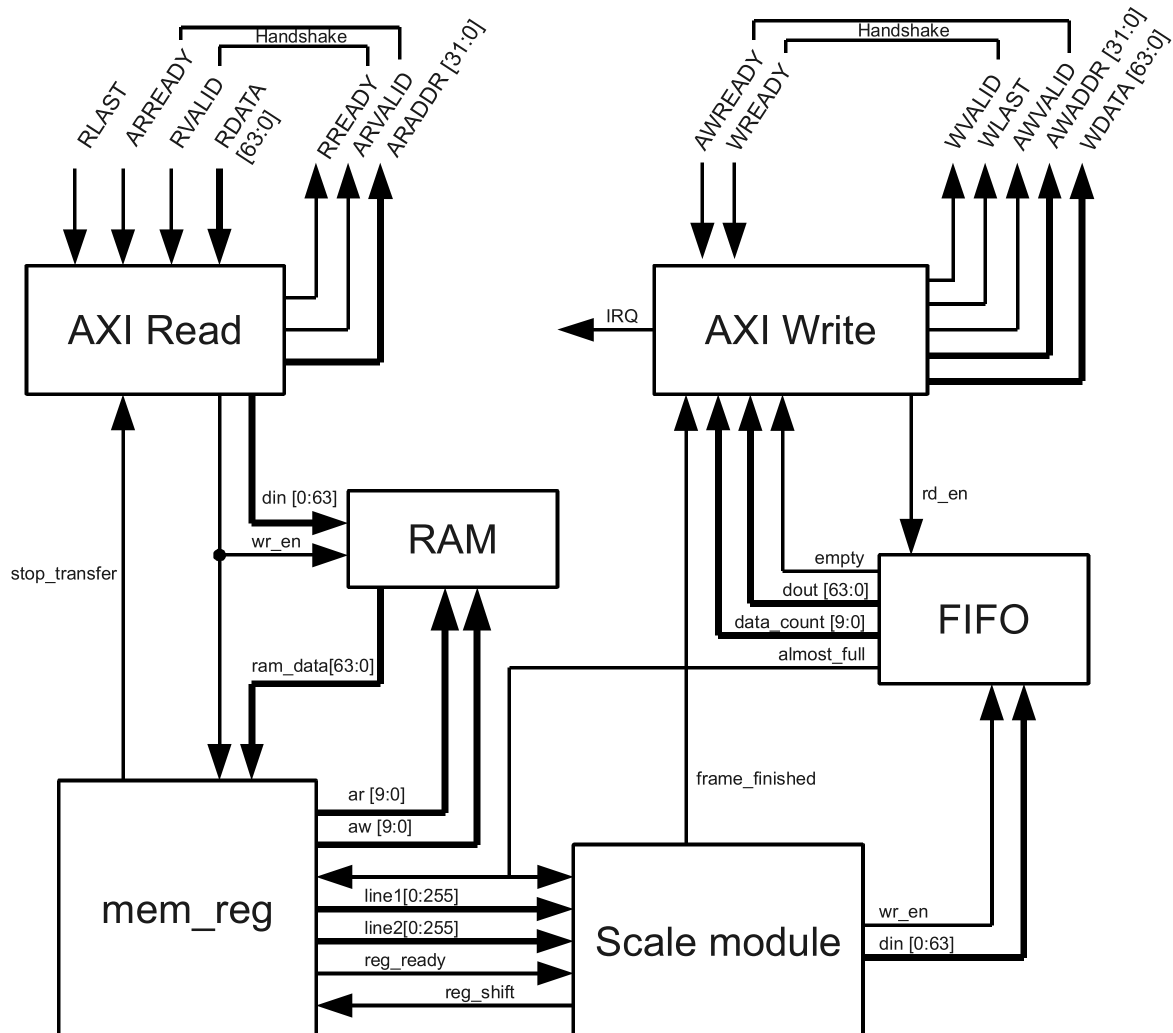


Figure 23: Video scaler system.

Some of the signals are not described before. `line1` and `line2` are the internal registers that mem_reg module handles. `ar` and `aw` are the addresses the RAM use for reading and writing. `frame_finished` is a signal asserted by the scale module when it is finished scaling the output frame. The `IRQ` signal is the signal that tells the APB module that the frame is finished written to the memory. The `data_count` signal from the FIFO holds

the amount of data packets in the FIFO. The AXI Write module only starts sending data over the AXI bus when the FIFO contains a whole burst. The reason for this is that the AXI bus should not be kept unnecessarily busy. A burst length of eight is used in this system.

6.5.1 Implementing system on the FPGA

The video scaler system was first tested with the verilog testbench, see section 5, and then implemented on an FPGA. The FPGA used was Xilinx Virtex-5 XC5VLX330T. Instead of the Mali core that ARM Trondheim usually implements in their FPGA system, the video scaler replaced this core. An overview of the implemented system on the FPGA is shown in figure 24.

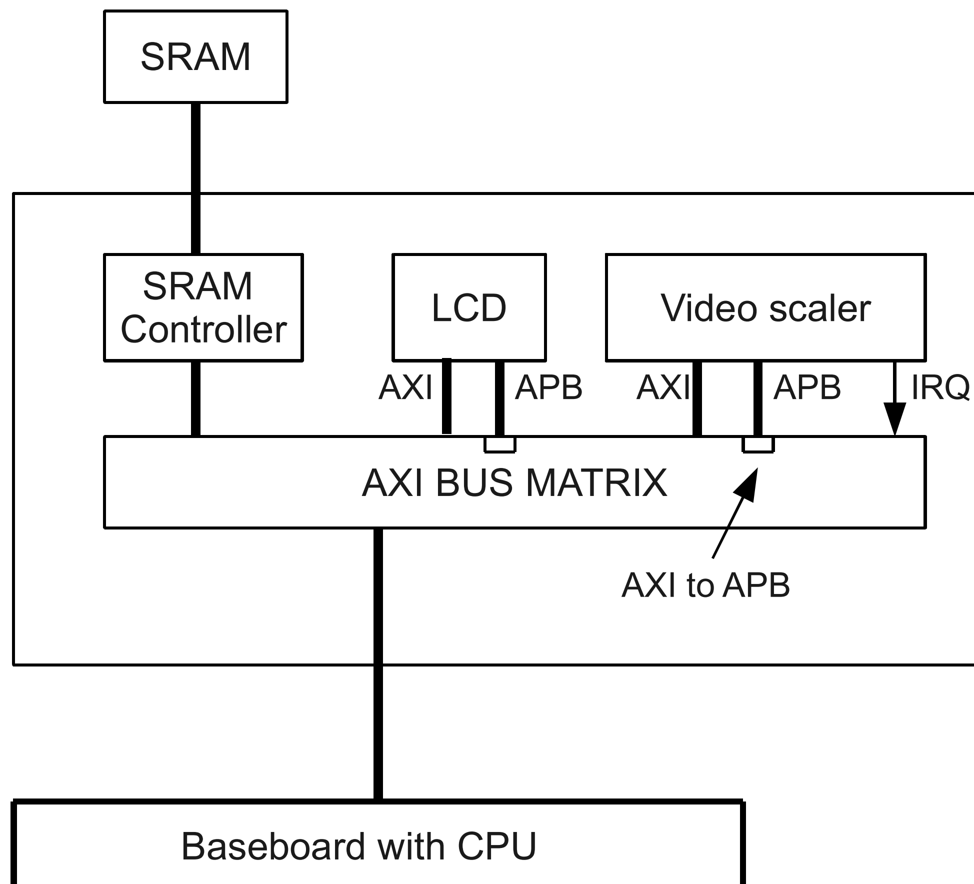


Figure 24: Overview of the system implemented on the FPGA.

Both the CPU and the Video Scaler have access to the SRAM through the Bus Matrix (and the SRAM controller). The Bus Matrix also contain AXI to APB bridges, since there is no dedicated APB bus going from the CPU baseboard up to the FPGA board. The LCD

is not used in this implementation. The video scaler also has its own interrupt channel (IRQ).

A program called *tb_util* was loaded in to the CPU and used to test the video scaler. By using this program it was possible to load the test set-up in to a configuration file, *config.txt*. This *config.txt* file is the same file that is used in the verilog testbench, but there is no option of setting read and write latency and outstanding transactions. First, the input frame is written to the SRAM via the AXI interface, then several APB writes is performed to configure and start the video scaler. When the video scaler has started scaling, the CPU waits for the interrupt from the video scaler. When the CPU receives the interrupt, it writes the scaled frame from SRAM to an output hex file. To test if the FPGA implementation scaled correctly it was compared with the output of the simulation, that was already tested to be correct. A *diff* command [5] was used to compare the output hex file that the FPGA produced with the expected result. The FPGA implementation worked for both scaling factors 1.25 and 1.875.

7 Results

This section first describes the performance of the two video scaling systems implemented on the FPGA. The two systems differs on the scaling factor, one has the factor 1.25, the other 1.875. Then follows a comparison of the implemented algorithms with well-known scaling algorithms. The output frame from scaling factor 1.5 is also compared with the algorithms, but this scaling factor was not implemented on the FPGA. Discussion of the result is done in section 8.

7.1 Performance in FPGA

The performance of the FPGA implementation of both scaling factors 1.25 and 1.875 are shown in tables 2 and 3, respectivly. The Xilinx log file for the two implemented video scalers is shown in appendix D.1 and D.2. The tables also list the throughput of bytes, pixels and frame. The FPGA is tested with a SD WIDE input frame (1024x576) with yuv444p format, and the FPGA was running with a fixed frequency of 50 MHz. To calculate output frames per second, equation 6 or 7 is used, depending on what pixel format is used.

$$\text{Throughput frames (format: yuv444p)} = \frac{50\text{ MHz}}{\text{output resolution} * 3} \quad (6)$$

$$\text{Throughput frames (format: yuv420p)} = \frac{50\text{ MHz}}{\text{output resolution} * 1.5} \quad (7)$$

LUTS	30,974 out of 207,360 (14%)
Registers	23,321 out of 207,360 (11%)
DSP48s	32 out of 192 (16%)
Clock frequency (fixed)	50 MHz
Throughput	0.28 bytes/clk
Throughput pixel (yuv444p)	0.84 pixel/clk
Throughput pixel (yuv420p)	0.42 pixel/clk
Throughput HD frames (yuv444p)	64.6 fps
Throughput HD frames (yuv420p)	129.2 fps

Table 2: Video scaler performance in FPGA with scaling factor 1.25.

Although the yuv420p format was not tested on the FPGA it is included in the figures. The reason for this is that the two formats, yuv420p and yuv444p, use exactly the same calculation method. But these values for yuv420p is only approximated values. The yuv420p format uses half the size of yuv444p.

LUTS	32,640 out of 207,360 (15%)
Registers	23,372 out of 207,360 (11%)
DSP48s	48 out of 192 (25%)
Clock frequency (fixed)	50 MHz
Throughput	0.25 bytes/clk
Throughput pixel (yuv444p)	0.75 pixel/clk
Throughput pixel (yuv420p)	0.38 pixel/clk
Throughput FullHD frames (yuv444p)	32.2 fps
Throughput FullHD frames (yuv420p)	64.3 fps

Table 3: Video scaler performance in FPGA with scaling factor 1.875.

7.2 Picture quality

To compare the picture quality of the two implemented algorithms a function in ImageMagick, compare [11], was used. This function can compare two pictures and calculate the difference between them depending on a metric. Six metrics were used for comparison, these were:

- MAE - mean absolute error, average channel error distance
- MSE - mean error squared, average of the channel error squared
- PSNR - peak signal to noise ratio
- RMSE - root mean squared
- AE - absolute number of different pixels
- AE -fuzz 1% - -fuzz factor to ignore pixels which only changed by a small amount.

The metrics that produce the best comparison results are PSNR and AE with fuzz factor of 1 %. The higher PSNR, and the lower AE -fuzz, the better scaling algorithm.

To do the comparison a picture had to be chosen. The Lenna picture [13] is a standard test image, so this was chosen for comparison. First, the picture had to be scaled down. This was performed both with GIMP Cubic and GIMP Linear. Secondly, it had to be scaled up again with different scaling algorithms. Scaling down with GIMP Linear and scaling up to the same resolution gave marginally better result, so the tables below use down scaling with GIMP Linear.

7.2.1 Scale factor 1.875

In table 4 a comparison of the different GIMP scaling algorithms [8] and the scaling algorithm used in the FPGA, with scaling factor 1.875 (SD2FullHD), is performed. Linear and Cubic is the same as bilinear and bicubic scaling. The Sinc algorithm is the Lanczos3

window filter [3]. GIMP none uses no interpolation, pixels are simply enlarged or removed, as they are when zooming.

Metric	GIMP none	GIMP Linear	GIMP Sinc	GIMP Cubic	SD2-FullHD
MAE	542.073	387.021	412.145	282.351	346.596
MSE	15.0218	4.65961	4.91951	1.91582	4.74291
PSNR	36.3975	41.4812	41.2455	45.3412	41.4043
RMSE	992.195	552.601	567.803	354.335	557.518
AE	885714	905683	909811	904984	880853
AE (1%)	242029	130011	173164	30018	109630

Table 4: Comparing FPGA implementation (SD2FullHD), with scaling factor 1.875, with GIMP scaling algorithms.

Table 4 shows as expected that bicubic interpolation is the best way to do up-scaling, and that no use of interpolation (GIMP none) produces the worst results. The three other algorithms have rather similar results. GIMP Linear and SD2FullHD produce slightly better results than GIMP Sinc. These differences need to be studied by viewing the output frame of the Lenna picture.

In table 5 a comparison of bicubic and bilinear scaling algorithms used in Corel Photoshop Pro [4] and the scaling algorithm used in the FPGA, with scaling factor 1.875 (SD2FullHD), is performed.

Metric	Corel bicubic	Corel bilinear	SD2-FullHD
MAE	325.919	408.102	346.596
MSE	3.93819	6.19559	4.74291
PSNR	42.2118	40.2439	41.4043
RMSE	508.025	637.203	557.518
AE	871883	890325	880853
AE (1%)	110285	172068	109630

Table 5: Comparing FPGA implementation (SD2FullHD), with scale factor 1.875, with Corel scaling algorithms.

Table 5 shows that Corel's bicubic and bilinear algorithms produce worse result than GIMP's algorithms. By studying the output picture and viewing the binary file in a hex-editor of both GIMP's and Corel's bilinear result, a likely reason for the different result could be how the two programs process colour. There is some difference in the colour, but both output pictures looks fine, nevertheless, GIMP's colour processing is established here to be better than Corel. This is also the case for the SD2FullHD algorithm, it does more precise colour processing.

7.2.2 Scale factor 1.25

In table 6 a comparison of the same GIMP scaling algorithms was performed, only with a scaling factor of 1.25 (SD2HD).

Metric	GIMP none	GIMP Linear	GIMP Sinc	GIMP Cubic	SD2HD
MAE	549.951	473.084	366.110	453.644	412.132
MSE	15.4370	7.41521	3.90219	6.13477	6.16828
PSNR	36.2791	39.4635	42.2516	40.2868	40.2631
RMSE	1005.81	697.105	505.698	634.068	635.797
AE	397492	404539	403169	405227	398642
AE (1%)	105597	92842	50112	96454	71283

Table 6: Comparing FPGA implementation, with scale factor 1.25, with GIMP scaling algorithms.

Unlike table 4, table 6 shows that GIMP Sinc algorithm is the best way to do up-scaling, when doing up scaling with a factor of 1.25. GIMP Cubic and SD2HD produces slightly better results than GIMP Linear, and no interpolation produces the worst result. When viewing the up-scaled pictures the results seem to correspond with the results in the table. It is actually difficult to determine which of GIMP Cubic and SD2HD is the better, SD2HD is bit more jagged than GIMP Cubic, but both seems to be slightly better than GIMP Linear.

7.2.3 Scale factor 1.5

In table 7 a comparison of the same GIMP scaling algorithms was performed, with a scaling factor of 1.5 (HD2FullHD).

Metric	GIMP none	GIMP Linear	GIMP Sinc	GIMP Cubic	HD2-FullHD
MAE	536.202	294.142	345.056	318.334	365.510
MSE	14.4038	4.90518	3.30025	2.80690	5.08882
PSNR	36.5800	41.2582	42.9793	43.6825	41.0986
RMSE	971.572	566.975	465.061	428.894	577.491
AE	888783	905995	905700	904767	882485
AE (1%)	245932	144828	98858	70372	136318

Table 7: Comparing FPGA implementation, with scale factor 1.5, with GIMP scaling algorithms.

Table 7 shows that GIMP Cubic algorithm is the best way to do up-scaling, when doing up scaling with a factor of 1.5. GIMP Sinc produces the next best result. GIMP Linear and HD2FullHD produces similar results, and no interpolation produces the worst result.

8 Discussion

This section includes a discussion of how HD2FullHD would have been implemented, some discussion of the results of the scaled pictures and the similarities to the bilinear interpolation method. The truncation of decimals in the implemented algorithms are discussed. In the last part of the section there is a discussion of the performance of the implemented modules, and then how the two video scaler systems could be merged in to one system.

It was desired to use a scaling factor that could handle all types of up-scaling first. During the work with the thesis the original request of handling arbitrary scaling factors were abandoned due to time constraints, and the focus was shifted to try to implement three static scaling factors. The main focus was to try to make HD or FullHD frames from a SD WIDE frame, scaling factors of 1.25 and 1.875, but scaling from HD to FullHD, scaling factor of 1.5, was also discussed.

8.1 Scale 1.5 (HD2FullHD)

The HD2FullHD algorithm with a scaling factor of 1.5 was not implemented in hardware due to time constraints. However, this algorithm would have been the easiest to implement of the three algorithms explained in section 3.2, *Video scaling algorithms for hardware.* The two other algorithms were given priority over scaling from HD to FullHD.

Since the two implemented algorithms are written so that it should be easy to rewrite code, the implementation of this algorithm (HD2FullHD) would not take long. The mem_reg module (Memory control module) would need to update the number of lines to 3, and the task function in the scaling module could have been the same as scaling factor 1.25 uses.

```
task avg_func;
  input [4:0] s1_r,s2_r,s3_r,s4_r; // s1+s2+s3+s4 = 16 // 4:0
  input [7:0] pix1_r,pix2_r,pix3_r,pix4_r; // pixel value
  output [7:0] po;
  reg [11:0] temp;
  begin
    if      (s1_r[4] == 1'b1) po = pix1_r;
    else if (s2_r[4] == 1'b1) po = pix2_r;
    else if (s3_r[4] == 1'b1) po = pix3_r;
    else if (s4_r[4] == 1'b1) po = pix4_r;
    else begin
      temp = ((pix1_r*s1_r[3:0]) + (pix2_r*s2_r[3:0]) + (pix3_r*s3_r[3:0]) +
           (pix4_r*s4_r[3:0]));
      po = temp[11:4];
    end
  end
endtask
```

The input s registers holds the dependency of the input pixel, pix register. In the case of HD2FullHD s would hold either 0, 4, 8 or 16, see figure 11 on page 13 for illustration of the dependencies. This could be changed to a fourth, so that s holds either 0, 1, 2 or 4, but that would lead to a small change in the task function used in Scale 1.25 module. A division of 4 and not 16 would have been the change, `po = temp[11:4];` to `po = temp[9:2];`. The input s registers could when be reduced to 3 bits. But by using exactly the same function as Scale 1.25 module, area can be saved.

The main difference between a Scale 1.5 module and the Scale 1.25 module would have been how to set the s registers that is the input to the task function, these are set regarding to the dependencies.

8.2 Smoothing effect on SD2FullHD

The algorithm used in the FPGA is nearly the same as bilinear interpolation, that for instance GIMP Linear uses, the main focus was to compare these two algorithms. Since the results of GIMP Linear and SD2FullHD were so similar, see section 7.2.1 and table 4, an illustrating comparison was needed.

After viewing the output frame from the SD2FullHD algorithm an issue with smoothing/interpolation could be seen. It seemed like it did not do any interpolation for some of the pixels. The frame did not look as smooth as predicted. When using very small parts of three input pixels and a large part of one input pixel, the effect of the three small part seemed to vanish. It is not used very much time to discuss this problem, but a quick fix was tried out. Trying to smooth the frame more than before, I simply used larger parts to interpolate with. For instance, when interpolating between the four parts, 1/64 p1, 7/64 p2, 7/64 p3 and 49/64 p4, the dependence of input pixel p1 surely has no effect since only one sixty fourth part depends on the output pixel. When one pixel was too dominant, I increased the dependencies of the other three pixels. By changing x and y in algorithm 1 on page 15 this could easily be implemented. The only difference to the algorithm was to change a 1 to a 3, and a 13 to a 11 on the x and y. For the four parts described earlier, the new interpolation would be 4/64 p1, 10/64 p2, 10/64 p3 and 30/64 p4. This lead to a smoother output frame.

Table 8 shows part of table 4 on page 42. The column that is added (SD2FullHD*) is the added smoothing effect on SD2FullHD. The table shows that the added smoothing effect on the SD2FullHD algorithm produces a better result.

Figure 25 shows the original SD2FullHD algorithms output of Lenna's eye. Figure 26 shows the effect of adding smoothness to SD2FullHD. Figure 27 shows the output of GIMP Linear, and figure 28 shows the original picture that was used for scaling. All pictures are zoomed in 400 %.

Metric	GIMP Linear	SD2-FullHD	SD2-FullHD*
MAE	387.021	346.596	339.835
MSE	4.65961	4.74291	4.45018
PSNR	41.4812	41.4043	41.6810
RMSE	552.601	557.518	540.039
AE	905683	880853	880883
AE (1%)	130011	109630	105740

Table 8: Effect of lacking smoothness.

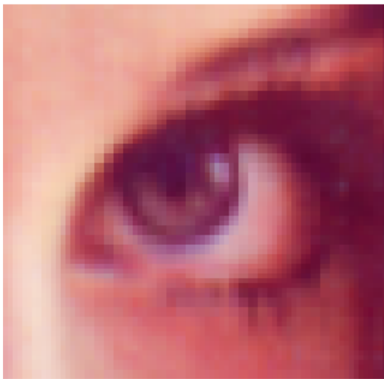


Figure 25: SD2FullHD accurate to dependencies (400% zoom).

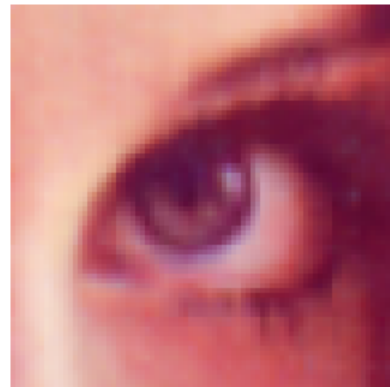


Figure 26: SD2FullHD with smoothing effect (400% zoom).

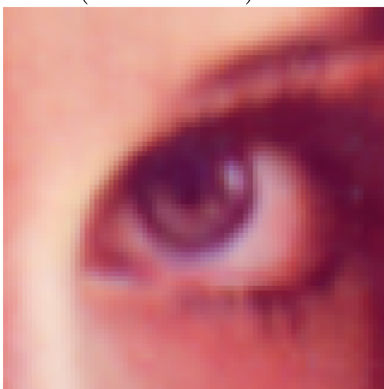


Figure 27: GIMP Linear output (400% zoom).

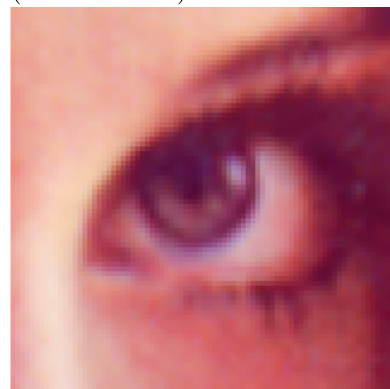


Figure 28: Original picture of Lenna's eye (400% zoom).

8.3 Similarities with bilinear interpolation

There are similarities between bilinear interpolation and my dependencies based algorithms. A comparison of parts of algorithm 1 on page 15 and bilinear interpolation is done. The part of the algorithm:

if x and y is odd **then**

$$po \Leftarrow (p1 * (8 - ((x+1)/2)) * (8 - ((y+1)/2)) + p2 * ((x+1) * 4 - ((x+1)/2) * ((y+1)/2)) + p3 * ((y+1) * 4 - ((x+1)/2) * ((y+1)/2)) + p4 * ((x+1)/2) * ((y+1)/2))/64;$$

end if

By replacing $(x+1)/2$ with x' and $(y+1)/2$ with y' the equation can be rewritten to:

if x and y is odd **then**

$$po \Leftarrow (p1 * (8 - x') * (8 - y') + p2 * x'(8 - y') + p3 * y'(8 - x') + p4 * (x' * y')/64;$$

end if

This part of the algorithm looks almost the same as the equation used for bilinear interpolation, equation 5 on page 5, also shown below.

$$f(x,y) = f(0,0)\,(1-x)(1-y) + f(1,0)\,x(1-y) + f(0,1)\,(1-x)y + f(1,1)xy$$

The main difference between the bilinear interpolation and my dependencies based algorithms is that bilinear uses a lot more smoothing by using all four surrounding pixel values to calculate an output pixel more often. The algorithms made for hardware in this project only uses all four surrounding input pixels when the output pixel is dependent on them, see section 3.2. As a consequence, these algorithms may not produce such a smooth picture as bilinear interpolation produces. On the other hand, the algorithms will not produce as much blur as bilinear interpolation.

In the results it was discussed a lot about smoothness, but as it was just stated, the algorithms made for hardware would not produce as blurry pictures as bilinear interpolation. To illustrate this a picture with text is used. Figure 29 shows a part of the ASCII characters that is going to be scaled with the factors 1.875, 1.5 and 1.25.

Dec	Hx	Oct	Char
0	0	000	NUL (null)
1	1	001	SOH (start of heading)

Figure 29: Part of the ASCII characters, source picture.

Figure 30 shows how the ASCII characters appear after scaling the source picture with a factor of 1.875. The bilinear appear much more blurry than the SD2FullHD algorithm.

Dec Hx Oct Char
0 0 000 NUL (null)
1 1 001 SOH (start of heading)

Dec Hx Oct Char
0 0 000 NUL (null)
1 1 001 SOH (start of heading)

Figure 30: ASCII characters scaled up with a factor of 1.875, bilinear interpolation (top) and SD2FullHD (bottom).

In the same way as scaling factor 1.875, figure 31 shows that the bilinear interpolation is more blurry than the HD2FullHD algorithm. The difference is not as significant as with scaling factor 1.875.

Dec Hx Oct Char
0 0 000 NUL (null)
1 1 001 SOH (start of heading)

Dec Hx Oct Char
0 0 000 NUL (null)
1 1 001 SOH (start of heading)

Figure 31: ASCII characters scaled up with a factor of 1.5, bilinear interpolation (top) and SD2FullHD (bottom).

Unlike the two previous scaling factors, it is difficult to see any difference between bilinear interpolation with scaling factor 1.25 and the SD2HD algorithm. This supports the similar results in section 7, only that SD2HD gave a bit better result. If you look very closely, there is a bit more shadowing on the letters in the picture scaled with bilinear interpolation.

Dec Hx Oct Char
0 0 000 NUL (null)
1 1 001 SOH (start of heading)

Dec Hx Oct Char
0 0 000 NUL (null)
1 1 001 SOH (start of heading)

Figure 32: ASCII characters scaled up with a factor of 1.25, bilinear interpolation (top) and SD2FullHD (bottom).

8.4 Truncation on the algorithms

All the algorithms implemented in hardware truncates all the decimal values when doing divison. The software implementation does the same.

To show a truncation example, the same example as in section 8.3 is used:

if x and y is odd **then**

$$po \Leftarrow (p1 * (8 - ((x+1)/2)) * (8 - ((y+1)/2))$$
$$+ p2 * ((x+1) * 4 - ((x+1)/2) * ((y+1)/2))$$
$$+ p3 * ((y+1) * 4 - ((x+1)/2) * ((y+1)/2))$$
$$+ p4 * ((x+1)/2) * ((y+1)/2))/64;$$

end if

Here you see that the output value po truncates all decimals when the result is divided by 64. Rounding up when the decimal value was 0.5 or more was applied in software. The effect of truncation versus rounding up was tested with the compare function used in section 7, *Results*. When using the compare function, the results were marginally better. Table 9 shows the results of truncation of decimals and rounding up when decimals are 0.5 or more.

The table shows that the effect of truncation has less than 1 % effect on the output frame. The effect of truncating decimal was not noticeable on the output picture. Since the effect is negligible, truncation is used in all the implemented algorithms.

8.5 Performance of implemented modules

All the modules used in the video scaler system were first synthesised with Synopsys Synplify Premier [16]. A summary of the Synplify log files are shown in the following tables. The log files for Scale 1.25 module, Scale 1.875 and mem_reg module can be found

Metric	SD2FullHD (truncation)	SD2FullHD (rounding up)
MAE	346.596	344.363
MSE	4.74291	4.47042
PSNR	41.4043	41.4249
RMSE	557.518	556.159
AE	880853	881102
AE (1%)	109630	108940

Table 9: Effect of truncation versus rounding up.

in appendix D.3, D.4 and D.5, respectively. The DSP48s is used for the multiplication in the Verilog task function.

Table 10 shows the performance of Scale 1.25 module.

LUTS	636 out of 207,360 (0%)
Registers	294 out of 207,360 (0%)
DSP48s	32 out of 192 (16%)
Clock frequency	347.3 MHz

Table 10: Scale 1.25 module performance.

Table 11 shows the performance of Scale 1.875 module.

LUTS	1,524 out of 207,360 (0%)
Registers	655 out of 207,360 (0%)
DSP48s	64 out of 192 (16%)
Clock frequency	343.9 MHz

Table 11: Scale 1.875 module performance.

Table 12 shows the performance of mem_reg module.

Table 12 shows the mem_reg module that is used with Scale 1.875 module. The result of the mem_reg module used with Scale 1.25 module is nearly the same, but uses a couple less registers and LUTS.

The three modules AXI Write, AXI Read and APB were also synthesised. These modules use a very small area, and performance frequency is well over the mem_reg module.

As mention in the results in section 7, the FPGA had a fixed performance frequency of 50 MHz, but in an ASIC design the frequency would be a lot higher. As shown in the tables in this section, the bottleneck of the system is certainly not the modules implemented for the video scaler.

LUTS	172 out of 207,360 (0%)
Registers	577 out of 207,360 (0%)
Clock frequency	199.6 MHz

Table 12: Mem_reg module performance.

8.6 Merging the video scaler systems

The time constraints on the thesis led to making two video scaler systems, one for scaling factor 1.25, one for scaling factor 1.875. The idea when the thesis started was to make a video scaler system that could choose between the different scaling factors. To merge the two mem_reg modules in to one would not be any problem since the two modules are nearly the same, but to merge the scaling modules together would be a more complicated assignment. It would be easy if we could choose between the modules to use, but since both modules use similar multiplication methods it would have been practical to save area by using the same multipliers. This would have been to time consuming with regards to the time constraints set on this thesis.

A scale module with up-scaling of 1.5 would not have been any problem to include in the merged video scaler since this scaling factor would have had a much simpler structure than the two other scaling factors already implemented.

In section 8.3 the similarities with bilinear interpolation is described. Parts of the scaling algorithms implemented in hardware use nearly the same calculation method as bilinear interpolation. By doing small modifications it could also have incorporated pure bilinear scaling. An option of switching between bilinear interpolation and the implemented algorithms would then be possible. This could be practical since bilinear produces smoother frames, but if the frame becomes too blurry with bilinear interpolation due to too much smoothing of sharp edges or text, a switch to the implemented algorithms would be possible. Then the video scaler system would have support for both sharp edge frames, such as text and maybe animated movies, and frame that is not edge sensitive, such as ordinary television viewing. To confirm the assumptions made in this section a real-time test on the FPGA that transfers the scaled frame to a TV would be necessary.

9 Conclusion

A set of algorithms developed by the author was implemented successfully on an FPGA. The implemented algorithms were compared with well-known algorithms typically used in video scalers, such as bilinear interpolation, bicubic interpolation and lanczos3 window filter. The main focus was to compare the implemented algorithms with bilinear interpolation since the two have some similarities. The algorithms seem to suit edge sensitive scaling better than bilinear. The output frame produce a bit more jagging, but is less blurry than bilinear.

Three scaling factors are compared, 1.25, 1.5 and 1.875. For scaling factor 1.25, the lanczos3 filter produced the best results, followed by similar results from bicubic and SD2HD. For scaling factor 1.5, the bicubic interpolation produced the best results, followed by the next best results from lanczos3, and similar results from bilinear and HD2FullHD. The last scaling factor, 1.875, showed that bicubic interpolation produced the best results, followed by bilinear and SD2FullHD with slightly better results than lanczos3.

The video scaler system performance was sufficient to run the video scaler in real time with live video output, but due to time constraints this was not tested.

Graphics scaling algorithms were written in software and compared. One algorithm was developed that produced good results, however, there were doubts whether the algorithm produced to much blur to suit graphics scaling.

10 Future work

I recommend that ARM Trondheim continue the video scaler project as a future master thesis. Then it would be possible to use all the results from this thesis and realise a video scaler system that can be tested in real time.

I suggest that the video scaler is implemented using bilinear interpolation and the algorithms implemented herein. In addition, the video scaler could be implemented to handle computer graphics scaling, or a independent graphics scaler can be implemented. Some research on this topic is included in this thesis, however, the implementation of such scaling is not described nor implemented.

Also, scaling from VGA resolution to FullHD resolution could be a smart function of a video scaler. By doing graphics up-scaling like for instance Hq3x you will get the resolution 1920x1440, when it is possible to scale down the vertical height with a factor of 0.75. This would certainly produce a nice output picture on a FullHD TV.

References

[1] ARM information center. http://infocenter.arm.com/help/index.jsp? topic=/com.arm.doc.set.amba/index.html.

[2] Bilinear interpolation - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Bilinear_interpolation.

[3] Claude E. Duchon. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology*, pages 1016–1022, 1979.

[4] Corel - photo editing - photos. http://www.corel.com/.

[5] diff(1): find diff between two files - linux man page. http://linux.die.net/man/1/diff.

[6] EasyBMP Cross-Platform windows BMP library: Home. http://easybmp.sourceforge.net/.

[7] Encoding parameters of digital television for studios, recommendation ITU-R BT.601-4, 1994.

[8] GIMP - the GNU image manipulation program. http://www.gimp.org/.

[9] GPU gems - chapter 25. GPU image processing in apple's motion. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter25.html.

[10] HiEnd3D. http://web.archive.org/web/20080208215126/http://www.hiend3d.com/hq3x.html.

[11] ImageMagick: command-line tools: Compare. http://www.imagemagick.org/script/compare.php.

[12] John Watkinson. *The MPEG Handbook*. Focal Press, second edition edition, 2004.

[13] Lenna - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Lenna.

[14] Pixel art scaling algorithms - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Pixel_art_scaling_algorithms.

[15] Scale2x. http://scale2x.sourceforge.net/algorithm.html.

[16] Synopsys - synplify premier. http://www.synopsys.com/Tools/Implementation /FPGAImplementation/FPGASynthesis/Pages/SynplifyPremier.aspx.

[17] Tech-Algorithm.com ˜ nearest neighbor image scaling. http://tech-algorithm.com/articles/nearest-neighbor-image-scaling/.

[18] Synopsys VCS. http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx.

[19] Welcome to FOURCC.org - YUV to RGB conversion. http://www.fourcc.org/fccyvrgb.php.

A C++ code

A.1 Algorithms for SD/HD/FullHD up-scaling

```
1
2  #include "EasyBMP_DataStructures.h"
3  #include "EasyBMP_BMP.h"
4
5  struct YUV{
6      double Y,U,V;
7  };
8  struct RGB{
9      double R,G,B;
10 };
11
12 YUV RGB2YUV(RGBApixel In);
13 RGB YUV2RGB(YUV In);
14
15 YUV funcModulo2(YUV In1, YUV In2, int x);
16 YUV funcOdd(YUV In0, YUV In1, YUV In2, YUV In3, int x, int y);
17 YUV avg4_12(YUV In1, YUV In2);
18 YUV avg8_8(YUV In1, YUV In2);
19 YUV avg1_3_3_9(YUV In1, YUV In2, YUV In3, YUV In4);
20 YUV avg2_2_6_6(YUV In1, YUV In2, YUV In3, YUV In4);
21 YUV avg4_4_4_4(YUV In1, YUV In2, YUV In3, YUV In4);
22
23 void SD2HD(BMP ImageIn, BMP &ImageOut)
24 {
25     //---------SD--------//        //--------------HD-------------//
26     // p1 | p2 | p3 | p4 //        // po1 | po2 | po3 | po4 | po5 //
27     // p5 | p6 | p7 | p8 //  --->  // po6 | po7 | po8 | po9 | po10//
28     // p9 | p10| p11| p12//        // po11| po12| po13| po14| po15//
29     // p13| p14| p15| p16//        // po16| po17| po18| po19| po20//
30     //-------------------//        // po21| po22| po23| po24| po25//
31                                    //-----------------------------//
32
33     int NewWidth = (int) (ImageIn.TellWidth() * 1.25);
34     int NewHeight = (int) (ImageIn.TellHeight() * 1.25);
35
36     int xo, yo = 0;
37
38     ImageOut.SetSize( NewWidth, NewHeight );
39
40     if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
41     else ImageOut.SetBitDepth(24);
42
43     for (int j=0 ; j < ImageIn.TellHeight(); j=j+4)
44     {
45      for (int i=0 ; i < ImageIn.TellWidth(); i=i+4)
46      {
```

```
47          // Safety if picture is not a factor of 4
48          if (4 <= ImageIn.TellWidth()-i &&
49              4 <= ImageIn.TellHeight()-j)
50          {
51              YUV p[17], po[26];
52
53              for (int x=1; x<17; x++)
54              {
55                  p[x] = RGB2YUV(*ImageIn(i+(x-1)%4,j+(x-1)/4));
56              }
57
58              // Calculate average values (See avg-functions)
59              po[1] = p[1]; po[5] = p[4]; po[21] = p[13]; po[25] = p[16];
60
61              po[2] = avg4_12(p[1],p[2]);
62              po[4] = avg4_12(p[4],p[3]);
63              po[6] = avg4_12(p[1],p[5]);
64              po[10] = avg4_12(p[4],p[8]);
65              po[16] = avg4_12(p[13],p[9]);
66              po[20] = avg4_12(p[16],p[12]);
67              po[22] = avg4_12(p[13],p[14]);
68              po[24] = avg4_12(p[16],p[15]);
69
70              po[3] = avg8_8(p[2],p[3]);
71              po[11] = avg8_8(p[5],p[9]);
72              po[15] = avg8_8(p[8],p[12]);
73              po[23] = avg8_8(p[14],p[15]);
74
75              po[7] = avg1_3_3_9(p[1],p[2],p[5],p[6]);
76              po[9] = avg1_3_3_9(p[4],p[3],p[8],p[7]);
77              po[17] = avg1_3_3_9(p[13],p[14],p[9],p[10]);
78              po[19] = avg1_3_3_9(p[16],p[12],p[15],p[11]);
79
80              po[8] = avg2_2_6_6(p[2],p[3],p[6],p[7]);
81              po[12] = avg2_2_6_6(p[5],p[9],p[6],p[10]);
82              po[14] = avg2_2_6_6(p[8],p[12],p[7],p[11]);
83              po[18] = avg2_2_6_6(p[14],p[15],p[10],p[11]);
84
85              po[13] = avg4_4_4_4(p[6],p[7],p[10],p[11]);
86
87              RGB RGBpo[26];
88              for (int x=1; x < 26; x++)
89              {
90                  RGBpo[x] = YUV2RGB(po[x]);
91              }
92
93              xo = (int)i*1.25; // Scale factor 1.25
94              yo = (int)j*1.25;
95
96              for (int x=0; x<5; x++)
97              {
```

```
98                      for (int y = 0; y<5; y++)
99                      {
100                         ImageOut(xo+x,yo+y)->Red = (char) RGBpo[x+y*5+1].R;
101                         ImageOut(xo+x,yo+y)->Green = (char) RGBpo[x+y*5+1].G;
102                         ImageOut(xo+x,yo+y)->Blue = (char) RGBpo[x+y*5+1].B;
103                     }
104                 }
105             }
106       }
107       cout << j << "␣/␣" << ImageIn.TellHeight() << endl;
108      }
109
110 }
111
112 void HD2FullHD(BMP ImageIn, BMP &ImageOut)
113 {
114     //----HD---//           //------FullHD-----//
115     // p1 | p2 //   x1.5    // po1 | po2 | po3 //
116     // p3 | p4 //   --->    // po4 | po5 | po6 //
117     //---------//           // po7 | po8 | po9 //
118                             //-----------------//
119
120     int NewWidth = (int) (ImageIn.TellWidth() * 1.5);
121     int NewHeight = (int) (ImageIn.TellHeight() * 1.5);
122
123     int xo, yo = 0;
124
125     ImageOut.SetSize( NewWidth, NewHeight );
126
127     if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
128     else ImageOut.SetBitDepth(24);
129
130     for (int j=0 ; j < ImageIn.TellHeight(); j=j+2)
131     {
132      for (int i=0 ; i < ImageIn.TellWidth(); i=i+2)
133      {
134          // Safety if picture is not a factor of 2
135          if (2 <= ImageIn.TellWidth()-i &&
136              2 <= ImageIn.TellHeight()-j)
137          {
138              YUV p[5], po[10];
139
140              for (int x=1; x<5; x++)
141              {
142                  p[x] = RGB2YUV(*ImageIn(i+(x-1)%2,j+(x-1)/2));
143              }
144
145              // Calculate average values (See avg-functions)
146              po[1] = p[1]; po[3] = p[2]; po[7] = p[3]; po[9] = p[4];
147
148              po[2] = avg8_8(p[1],p[2]);
```

```
149                 po[4] = avg8_8(p[1],p[3]);
150                 po[6] = avg8_8(p[2],p[4]);
151                 po[8] = avg8_8(p[3],p[4]);
152 
153                 po[5] = avg4_4_4_4(p[1],p[2],p[3],p[4]);
154 
155                 RGB RGBpo[10];
156                 for (int x=1; x < 10; x++)
157                 {
158                     RGBpo[x] = YUV2RGB(po[x]);
159                 }
160 
161                 xo = (int)i*1.5; // Scale factor 1.5
162                 yo = (int)j*1.5;
163 
164                 for (int x=0; x<3; x++)
165                 {
166                     for (int y = 0; y<3; y++)
167                     {
168                         ImageOut(xo+x,yo+y)->Red = (char) RGBpo[x+y*3+1].R;
169                         ImageOut(xo+x,yo+y)->Green = (char) RGBpo[x+y*3+1].G;
170                         ImageOut(xo+x,yo+y)->Blue = (char) RGBpo[x+y*3+1].B;
171                     }
172                 }
173             }
174      }
175      cout << j << " / " << ImageIn.TellHeight() << endl;
176     }
177 
178 }
179 
180 void SD2FullHD(BMP ImageIn, BMP &ImageOut)
181 {
182     //----SD---//          //-FullHD-//
183     // p1 | p2 //  x1.875  //   po   // with regards of 4 input pixels
184     // p3 | p4 //   --->   //--------//
185     //---------//
186 
187     /* Algorithm SD2FullHD (8x8 -> 15x15)
188      * for all po (x and y of input picture)
189      *   estimate average of all 4 souronding pixels by using the
           relationship:
190      *   po = p1*(16-x-y)/16 + p2*(x-x/16*y/16) + p3*(x-x/16*y/16)
191      *        p4*x/16*y/16;
192      *
193      *   when y = 4 and x = 4 symmetri is reached
194      *
195      * */
196 
197     int NewWidth = (int) (ImageIn.TellWidth() * 1.875);
198     int NewHeight = (int) (ImageIn.TellHeight() * 1.875);
```

```
199
200     YUV p[4],pout, p0;
201     p0.Y = 0; p0.U = 0; p0.V = 0;
202
203     ImageOut.SetSize( NewWidth, NewHeight );
204
205     if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
206     else ImageOut.SetBitDepth(24);
207
208     for (int j=0 ; j < ImageOut.TellHeight()/15; j++)
209     {
210      for (int i=0 ; i < ImageOut.TellWidth()/15; i++)
211      {
212          for (int y=0; y<15; y++)
213          {
214              for (int x=0; x<15; x++)
215              {
216                  p[0] = RGB2YUV(*ImageIn(i*8+x/2,j*8+y/2));
217                  p[1] = RGB2YUV(*ImageIn(i*8+(x/2)+1,j*8+y/2));
218                  p[2] = RGB2YUV(*ImageIn(i*8+x/2,j*8+(y/2)+1));
219                  p[3] = RGB2YUV(*ImageIn(i*8+(x/2)+1,j*8+(y/2)+1));
220
221                  if ( (x%2)==0 && (y%2)==0) pout = p[0];
222                  else if (y%2==0) pout = funcModulo2(p[0],p[1],x);
223                  //else if (y%2==0) pout = (p[0]*(8-(x+1/2)) +
                         p[1]*(x+1/2))/8;
224                  else if (x%2==0) pout = funcModulo2(p[0],p[2],y);
225                  //else if (x%2==0) pout = (p[0]*(8-(y+1/2)) +
                         p[2]*(y+1/2))/8;
226                  else pout = funcOdd(p[0], p[1], p[2], p[3], x, y);
227
228                  RGB RGBpout = YUV2RGB(pout);
229
230                  ImageOut(i*15+x,j*15+y)->Red = (char) RGBpout.R;
231                  ImageOut(i*15+x,j*15+y)->Green = (char) RGBpout.G;
232                  ImageOut(i*15+x,j*15+y)->Blue = (char) RGBpout.B;
233              }
234          }
235
236      }
237      cout << j << " / " << ImageOut.TellHeight()/15 << endl;
238     }
239
240 }
241
242 YUV funcModulo2(YUV In1, YUV In2, int x)
243 {
244     YUV Out;
245
246     Out.Y = (8-((x+1)/2))*In1.Y*0.125 + ((x+1)/2)*In2.Y*0.125;
247     Out.U = (8-((x+1)/2))*In1.U*0.125 + ((x+1)/2)*In2.U*0.125;
```

```
248     Out.V = (8-((x+1)/2))*In1.V*0.125 + ((x+1)/2)*In2.V*0.125;
249
250     return Out;
251 }
252
253 YUV funcOdd(YUV In0, YUV In1, YUV In2, YUV In3, int x, int y)
254 {
255     YUV Out;
256
257     Out.Y = ( ((x+1)/2)*((y+1)/2)*In3.Y +
258             ((y+1)*4 - ((x+1)/2)*((y+1)/2))*In2.Y +
259             ((x+1)*4 - ((x+1)/2)*((y+1)/2))*In1.Y +
260             (8-((x+1)/2))*(8-((y+1)/2))*In0.Y )/64;
261     Out.U = ( ((x+1)/2)*((y+1)/2)*In3.U +
262             ((y+1)*4 - ((x+1)/2)*((y+1)/2))*In2.U +
263             ((x+1)*4 - ((x+1)/2)*((y+1)/2))*In1.U +
264             (8-((x+1)/2))*(8-((y+1)/2))*In0.U )/64;
265     Out.V = ( ((x+1)/2)*((y+1)/2)*In3.V +
266             ((y+1)*4 - ((x+1)/2)*((y+1)/2))*In2.V +
267             ((x+1)*4 - ((x+1)/2)*((y+1)/2))*In1.V +
268             (8-((x+1)/2))*(8-((y+1)/2))*In0.V )/64;
269
270     return Out;
271 }
272
273 YUV avg4_12(YUV In1, YUV In2)
274 {
275     YUV Out;
276
277     Out.Y = (4*In1.Y + 12*In2.Y)/16;
278     Out.U = (4*In1.U + 12*In2.U)/16;
279     Out.V = (4*In1.V + 12*In2.V)/16;
280
281     return Out;
282 }
283 YUV avg8_8(YUV In1, YUV In2)
284 {
285     YUV Out;
286 /*
287     Out.Y = (8*In1.Y + 8*In2.Y)/16;
288     Out.U = (8*In1.U + 8*In2.U)/16;
289     Out.V = (8*In1.V + 8*In2.V)/16;
290 */
291     Out.Y = (In1.Y + In2.Y)/2;
292     Out.U = (In1.U + In2.U)/2;
293     Out.V = (In1.V + In2.V)/2;
294
295     return Out;
296 }
297 YUV avg1_3_3_9(YUV In1, YUV In2, YUV In3, YUV In4)
298 {
```

```
299     YUV Out;
300     Out.Y = (1*In1.Y + 3*In2.Y + 3*In3.Y + 9*In4.Y)/16;
301     Out.U = (1*In1.U + 3*In2.U + 3*In3.U + 9*In4.U)/16;
302     Out.V = (1*In1.V + 3*In2.V + 3*In3.V + 9*In4.V)/16;
303
304     return Out;
305 }
306 YUV avg2_2_6_6(YUV In1, YUV In2, YUV In3, YUV In4)
307 {
308     YUV Out;
309     Out.Y = (2*In1.Y + 2*In2.Y + 6*In3.Y + 6*In4.Y)/16;
310     Out.U = (2*In1.U + 2*In2.U + 6*In3.U + 6*In4.U)/16;
311     Out.V = (2*In1.V + 2*In2.V + 6*In3.V + 6*In4.V)/16;
312
313     return Out;
314 }
315 YUV avg4_4_4_4(YUV In1, YUV In2, YUV In3, YUV In4)
316 {
317     YUV Out;
318 /*
319     Out.Y = (4*In1.Y + 4*In2.Y + 4*In3.Y + 4*In4.Y)/16;
320     Out.U = (4*In1.U + 4*In2.U + 4*In3.U + 4*In4.U)/16;
321     Out.V = (4*In1.V + 4*In2.V + 4*In3.V + 4*In4.V)/16;
322 */
323     Out.Y = (In1.Y + In2.Y + In3.Y + In4.Y)/4;
324     Out.U = (In1.U + In2.U + In3.U + In4.U)/4;
325     Out.V = (In1.V + In2.V + In3.V + In4.V)/4;
326
327     return Out;
328 }
329
330
331 YUV RGB2YUV(RGBApixel In)
332 {
333   YUV Out; // legge inn (int), raskere?
334
335   /* Factors for implementin in HW */
336   // Out.Y = In.Red/4 + In.Green/2 + In.Blue/4;
337   // Out.U = (In.Blue - Out.Y)/2;
338   // Out.V = (In.Red - Out.Y)/2;
339
340   /* Original transform factors */
341   Out.Y = 0.299*In.Red + 0.587*In.Green + 0.114*In.Blue;
342   Out.U = (In.Blue - Out.Y)*0.565;
343   Out.V = (In.Red - Out.Y)*0.713;
344
345   return Out;
346 }
347
348 RGB YUV2RGB(YUV In)
349 {
```

```
350   RGB Out;
351
352   /* Factors for implementin in HW */
353   // Out.R = 2*In.V + In.Y;
354   // Out.G = In.Y - In.U/2 - In.V/2;
355   // Out.B = In.Y + 2*In.U;
356
357   /* Original transform faktors */
358   Out.R = 1.403*In.V + In.Y;
359   Out.G = In.Y - 0.344*In.U - 0.714*In.V;
360   Out.B = In.Y + 1.77*In.U;
361
362   return Out;
363 }
```

A.2 Algorithms for graphic scaling (RGB)

```
void simpleDoubleAlgorithm(BMP ImageIn, BMP &ImageOut)
{
    int NewWidth = (int) (ImageIn.TellWidth() * 2);
    int NewHeight = (int) (ImageIn.TellHeight() * 2);

    ImageOut.SetSize( NewWidth, NewHeight );

    if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
    else ImageOut.SetBitDepth(24);

    for (int j=0 ; j < ImageOut.TellHeight() ; j++)
    {
     for (int i=0 ; i < ImageOut.TellWidth() ; i++)
     {
        ImageOut(i,j)->Red = ImageIn(i/2,j/2)->Red;
        ImageOut(i,j)->Blue = ImageIn(i/2,j/2)->Blue;
        ImageOut(i,j)->Green = ImageIn(i/2,j/2)->Green;
        ImageOut(i,j)->Alpha = ImageIn(i/2,j/2)->Alpha;
     }
    }
}

void scale2xAlgorithm(BMP ImageIn, BMP &ImageOut)
{
    int NewWidth = (int) ( ImageIn.TellWidth() * 2);
    int NewHeight = (int) ( ImageIn.TellHeight() * 2);

    ImageOut.SetSize( NewWidth, NewHeight );

    if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
    else ImageOut.SetBitDepth(24);

    for (int j=0 ; j < ImageIn.TellHeight() ; j++)
    {
     for (int i=0 ; i < ImageIn.TellWidth() ; i++)
     {

          if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
             && j != ImageIn.TellHeight()-1
             && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
             && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
             && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
             && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
             && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
             && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue)
          {

              ImageOut(i*2,j*2)->Red = ImageIn(i-1,j)->Red ==
                  ImageIn(i,j-1)->Red ?
```

```
49                 ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
50           ImageOut(i*2+1,j*2)->Red = ImageIn(i,j-1)->Red ==
                ImageIn(i+1,j)->Red ?
51                 ImageIn(i+1,j)->Red : ImageIn(i,j)->Red;
52           ImageOut(i*2,j*2+1)->Red = ImageIn(i-1,j)->Red ==
                ImageIn(i,j+1)->Red ?
53                 ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
54           ImageOut(i*2+1,j*2+1)->Red = ImageIn(i,j+1)->Red ==
                ImageIn(i+1,j)->Red ?
55                 ImageIn(i+1,j)->Red : ImageIn(i,j)->Red;
56        }
57        else
58        {
59           ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;
60           ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;
61           ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;
62           ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;
63        }
64
65        if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
66           && j != ImageIn.TellHeight()-1
67           && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
68           && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
69           && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
70           && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
71           && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
72           && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue )
73        {
74           ImageOut(i*2,j*2)->Green = ImageIn(i-1,j)->Green ==
                ImageIn(i,j-1)->Green ?
75                 ImageIn(i-1,j)->Green : ImageIn(i,j)->Green;
76           ImageOut(i*2+1,j*2)->Green = ImageIn(i,j-1)->Green ==
                ImageIn(i+1,j)->Green ?
77                 ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
78           ImageOut(i*2,j*2+1)->Green = ImageIn(i-1,j)->Green ==
                ImageIn(i,j+1)->Green ?
79                 ImageIn(i-1,j)->Green : ImageIn(i,j)->Green;
80           ImageOut(i*2+1,j*2+1)->Green = ImageIn(i,j+1)->Green ==
                ImageIn(i+1,j)->Green ?
81                 ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
82        }
83        else
84        {
85           ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
86           ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
87           ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
88           ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
89        }
90
91
92        if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
```

```
93              && j != ImageIn.TellHeight()-1
94              && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
95              && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue
96              && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
97              && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
98              && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
99              && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red )
100          {
101              ImageOut(i*2,j*2)->Blue = ImageIn(i-1,j)->Blue ==
                     ImageIn(i,j-1)->Blue ?
102                  ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;
103              ImageOut(i*2+1,j*2)->Blue = ImageIn(i,j-1)->Blue ==
                     ImageIn(i+1,j)->Blue ?
104                  ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;
105              ImageOut(i*2,j*2+1)->Blue = ImageIn(i-1,j)->Blue ==
                     ImageIn(i,j+1)->Blue ?
106                  ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;
107              ImageOut(i*2+1,j*2+1)->Blue = ImageIn(i,j+1)->Blue ==
                     ImageIn(i+1,j)->Blue ?
108                  ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;
109          }
110          else
111          {
112             ImageOut(2*i,2*j)->Blue = ImageIn(i,j)->Blue;
113             ImageOut(2*i+1,2*j)->Blue = ImageIn(i,j)->Blue;
114             ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
115             ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;
116          }
117
118      }
119     }
120 }
121
122 void scale3xAlgorithm(BMP ImageIn, BMP &ImageOut)
123 {
124     int NewWidth = (int) ( ImageIn.TellWidth() * 3);
125     int NewHeight = (int) ( ImageIn.TellHeight() * 3);
126
127     ImageOut.SetSize( NewWidth, NewHeight );
128
129     if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
130     else ImageOut.SetBitDepth(24);
131
132     for (int j=0 ; j < ImageIn.TellHeight() ; j++)
133     {
134      for (int i=0 ; i < ImageIn.TellWidth() ; i++)
135      {
136          if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
137             && j != ImageIn.TellHeight()-1
138             && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
139             && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
```

```
140              && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
141              && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
142              && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
143              && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue)
144           {
145               ImageOut(i*3,j*3)->Red = ImageIn(i-1,j)->Red ==
                      ImageIn(i,j-1)->Red ?
146                    ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
147
148               ImageOut(i*3+1,j*3)->Red =
149                 (ImageIn(i-1,j)->Red == ImageIn(i,j-1)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i+1,j-1)->Red)
150                 || (ImageIn(i,j-1)->Red == ImageIn(i+1,j)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i-1,j-1)->Red)
151                 ? ImageIn(i,j-1)->Red : ImageIn(i,j)->Red;
152
153               ImageOut(i*3+2,j*3)->Red = ImageIn(i,j-1)->Red ==
                      ImageIn(i+1,j)->Red ?
154                    ImageIn(i+1,j)->Red : ImageIn(i,j)->Red;
155
156               ImageOut(i*3,j*3+1)->Red =
157                 (ImageIn(i-1,j)->Red == ImageIn(i,j-1)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i-1,j+1)->Red)
158                 || (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i-1,j-1)->Red)
159                 ? ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
160
161               ImageOut(i*3+1,j*3+1)->Red = ImageIn(i,j)->Red;
162
163               ImageOut(i*3+2,j*3+1)->Red =
164                 (ImageIn(i,j-1)->Red == ImageIn(i+1,j)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i+1,j+1)->Red)
165                 || (ImageIn(i,j+1)->Red == ImageIn(i+1,j)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i+1,j-1)->Red)
166                 ? ImageIn(i+1,j)->Red : ImageIn(i,j)->Red;
167
168               ImageOut(i*3,j*3+2)->Red = ImageIn(i-1,j)->Red ==
                      ImageIn(i,j+1)->Red ?
169                    ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
170
171               ImageOut(i*3+1,j*3+2)->Red =
172                 (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i+1,j+1)->Red)
173                 || (ImageIn(i,j+1)->Red == ImageIn(i+1,j)->Red &&
                        ImageIn(i,j)->Red != ImageIn(i,j+1)->Red)
174                 ? ImageIn(i,j+1)->Red : ImageIn(i,j)->Red;
175
176               ImageOut(i*3+2,j*3+2)->Red = ImageIn(i,j+1)->Red ==
                      ImageIn(i+1,j)->Red ?
177                    ImageIn(i+1,j)->Red : ImageIn(i,j)->Red;
178           }
```

```
179         else
180         {
181            ImageOut(3*i,3*j)->Red = ImageIn(i,j)->Red;
182            ImageOut(3*i+1,3*j)->Red = ImageIn(i,j)->Red;
183            ImageOut(3*i+2,3*j)->Red = ImageIn(i,j)->Red;
184            ImageOut(3*i,3*j+1)->Red = ImageIn(i,j)->Red;
185            ImageOut(3*i+1,3*j+1)->Red = ImageIn(i,j)->Red;
186            ImageOut(3*i+2,3*j+1)->Red = ImageIn(i,j)->Red;
187            ImageOut(3*i,3*j+2)->Red = ImageIn(i,j)->Red;
188            ImageOut(3*i+1,3*j+2)->Red = ImageIn(i,j)->Red;
189            ImageOut(3*i+2,3*j+2)->Red = ImageIn(i,j)->Red;
190         }
191         if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
192            && j != ImageIn.TellHeight()-1
193            && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
194            && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
195            && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
196            && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
197            && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
198            && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue)
199         {
200             ImageOut(i*3,j*3)->Green = ImageIn(i-1,j)->Green ==
                    ImageIn(i,j-1)->Green ?
201                  ImageIn(i-1,j)->Green : ImageIn(i,j)->Green;
202
203             ImageOut(i*3+1,j*3)->Green =
204               (ImageIn(i-1,j)->Green == ImageIn(i,j-1)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i+1,j-1)->Green)
205               || (ImageIn(i,j-1)->Green == ImageIn(i+1,j)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i-1,j-1)->Green)
206               ? ImageIn(i,j-1)->Green : ImageIn(i,j)->Green;
207
208             ImageOut(i*3+2,j*3)->Green = ImageIn(i,j-1)->Green ==
                    ImageIn(i+1,j)->Green ?
209                  ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
210
211             ImageOut(i*3,j*3+1)->Green =
212               (ImageIn(i-1,j)->Green == ImageIn(i,j-1)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i-1,j+1)->Green)
213               || (ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i-1,j-1)->Green)
214               ? ImageIn(i-1,j)->Green : ImageIn(i,j)->Green;
215
216             ImageOut(i*3+1,j*3+1)->Green = ImageIn(i,j)->Green;
217
218             ImageOut(i*3+2,j*3+1)->Green =
219               (ImageIn(i,j-1)->Green == ImageIn(i+1,j)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i+1,j+1)->Green)
220               || (ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i+1,j-1)->Green)
221               ? ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
```

```
222
223             ImageOut(i*3,j*3+2)->Green = ImageIn(i-1,j)->Green ==
                    ImageIn(i,j+1)->Green ?
224                  ImageIn(i-1,j)->Green : ImageIn(i,j)->Green;
225
226             ImageOut(i*3+1,j*3+2)->Green =
227               (ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i+1,j+1)->Green)
228               || (ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green &&
                      ImageIn(i,j)->Green != ImageIn(i,j+1)->Green)
229               ? ImageIn(i,j+1)->Green : ImageIn(i,j)->Green;
230
231             ImageOut(i*3+2,j*3+2)->Green = ImageIn(i,j+1)->Green ==
                    ImageIn(i+1,j)->Green ?
232                  ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
233         }
234         else
235         {
236            ImageOut(3*i,3*j)->Green = ImageIn(i,j)->Green;
237            ImageOut(3*i+1,3*j)->Green = ImageIn(i,j)->Green;
238            ImageOut(3*i+2,3*j)->Green = ImageIn(i,j)->Green;
239            ImageOut(3*i,3*j+1)->Green = ImageIn(i,j)->Green;
240            ImageOut(3*i+1,3*j+1)->Green = ImageIn(i,j)->Green;
241            ImageOut(3*i+2,3*j+1)->Green = ImageIn(i,j)->Green;
242            ImageOut(3*i,3*j+2)->Green = ImageIn(i,j)->Green;
243            ImageOut(3*i+1,3*j+2)->Green = ImageIn(i,j)->Green;
244            ImageOut(3*i+2,3*j+2)->Green = ImageIn(i,j)->Green;
245         }
246
247         if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
248            && j != ImageIn.TellHeight()-1
249            && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
250            && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
251            && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
252            && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
253            && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
254            && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue )
255         {
256             ImageOut(i*3,j*3)->Blue = ImageIn(i-1,j)->Blue ==
                    ImageIn(i,j-1)->Blue ?
257                  ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;
258
259             ImageOut(i*3+1,j*3)->Blue =
260               (ImageIn(i-1,j)->Blue == ImageIn(i,j-1)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i+1,j-1)->Blue)
261               || (ImageIn(i,j-1)->Blue == ImageIn(i+1,j)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i-1,j-1)->Blue)
262               ? ImageIn(i,j-1)->Blue : ImageIn(i,j)->Blue;
263
264             ImageOut(i*3+2,j*3)->Blue = ImageIn(i,j-1)->Blue ==
                    ImageIn(i+1,j)->Blue ?
```

```
265                 ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;
266
267             ImageOut(i*3,j*3+1)->Blue =
268               (ImageIn(i-1,j)->Blue == ImageIn(i,j-1)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i-1,j+1)->Blue)
269               || (ImageIn(i-1,j)->Blue == ImageIn(i,j+1)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i-1,j-1)->Blue)
270               ? ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;
271
272             ImageOut(i*3+1,j*3+1)->Blue = ImageIn(i,j)->Blue;
273
274             ImageOut(i*3+2,j*3+1)->Blue =
275               (ImageIn(i,j-1)->Blue == ImageIn(i+1,j)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i+1,j+1)->Blue)
276               || (ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i+1,j-1)->Blue)
277               ? ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;
278
279             ImageOut(i*3,j*3+2)->Blue = ImageIn(i-1,j)->Blue ==
                   ImageIn(i,j+1)->Blue ?
280                 ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;
281
282             ImageOut(i*3+1,j*3+2)->Blue =
283               (ImageIn(i-1,j)->Blue == ImageIn(i,j+1)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i+1,j+1)->Blue)
284               || (ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue &&
                      ImageIn(i,j)->Blue != ImageIn(i,j+1)->Blue)
285               ? ImageIn(i,j+1)->Blue : ImageIn(i,j)->Blue;
286
287             ImageOut(i*3+2,j*3+2)->Blue = ImageIn(i,j+1)->Blue ==
                   ImageIn(i+1,j)->Blue ?
288                 ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;
289         }
290         else
291         {
292            ImageOut(3*i,3*j)->Blue = ImageIn(i,j)->Blue;
293            ImageOut(3*i+1,3*j)->Blue = ImageIn(i,j)->Blue;
294            ImageOut(3*i+2,3*j)->Blue = ImageIn(i,j)->Blue;
295            ImageOut(3*i,3*j+1)->Blue = ImageIn(i,j)->Blue;
296            ImageOut(3*i+1,3*j+1)->Blue = ImageIn(i,j)->Blue;
297            ImageOut(3*i+2,3*j+1)->Blue = ImageIn(i,j)->Blue;
298            ImageOut(3*i,3*j+2)->Blue = ImageIn(i,j)->Blue;
299            ImageOut(3*i+1,3*j+2)->Blue = ImageIn(i,j)->Blue;
300            ImageOut(3*i+2,3*j+2)->Blue = ImageIn(i,j)->Blue;
301         }
302
303     }
304    }
305
306 }
307
```

```
308 // Interpolation with RGB:
309 void rogers2xAlgorithm(BMP ImageIn, BMP &ImageOut)
310 {
311 
312     int NewWidth = (int) ( ImageIn.TellWidth() * 2);
313     int NewHeight = (int) ( ImageIn.TellHeight() * 2);
314 
315     ImageOut.SetSize( NewWidth, NewHeight );
316 
317     if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
318     else ImageOut.SetBitDepth(24);
319 
320     for (int j=0 ; j < ImageIn.TellHeight() ; j++)
321     {
322       for (int i=0 ; i < ImageIn.TellWidth() ; i++)
323       {
324          if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
325             && j != ImageIn.TellHeight()-1
326             && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
327             && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
328             && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
329             && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue
330             && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
331             && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green )
332          {
333              ImageOut(i*2,j*2)->Red =
334                (ImageIn(i-1,j-1)->Red + ImageIn(i,j-1)->Red +
335                ImageIn(i-1,j)->Red + 4*ImageIn(i,j)->Red)/8;
336              ImageOut(i*2+1,j*2)->Red =
337                (ImageIn(i,j-1)->Red + ImageIn(i+1,j-1)->Red +
338                4*ImageIn(i,j)->Red + ImageIn(i+1,j)->Red)/8;
339              ImageOut(i*2,j*2+1)->Red =
340                (ImageIn(i-1,j)->Red + 4*ImageIn(i,j)->Red +
341                ImageIn(i-1,j+1)->Red + ImageIn(i,j+1)->Red)/8;
342              ImageOut(i*2+1,j*2+1)->Red =
343                (4*ImageIn(i,j)->Red + ImageIn(i+1,j)->Red +
344                ImageIn(i,j+1)->Red + ImageIn(i+1,j+1)->Red)/8;
345          }
346          else
347          {
348             ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;
349             ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;
350             ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;
351             ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;
352          }
353 
354          if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
355             && j != ImageIn.TellHeight()-1
356             && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
357             && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue
358             && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
```

```
359               && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
360               && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
361               && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green )
362            {
363                ImageOut(i*2,j*2)->Blue =
364                  (ImageIn(i-1,j-1)->Blue + ImageIn(i,j-1)->Blue +
365                  ImageIn(i-1,j)->Blue + 4*ImageIn(i,j)->Blue)/8;
366                ImageOut(i*2+1,j*2)->Blue =
367                  (ImageIn(i,j-1)->Blue + ImageIn(i+1,j-1)->Blue +
368                  4*ImageIn(i,j)->Blue + ImageIn(i+1,j)->Blue)/8;
369                ImageOut(i*2,j*2+1)->Blue =
370                  (ImageIn(i-1,j)->Blue + 4*ImageIn(i,j)->Blue +
371                  ImageIn(i-1,j+1)->Blue + ImageIn(i,j+1)->Blue)/8;
372                ImageOut(i*2+1,j*2+1)->Blue =
373                  (4*ImageIn(i,j)->Blue + ImageIn(i+1,j)->Blue +
374                  ImageIn(i,j+1)->Blue + ImageIn(i+1,j+1)->Blue)/8;
375            }
376            else
377            {
378               ImageOut(2*i,2*j)->Blue = ImageIn(i,j)->Blue;
379               ImageOut(2*i+1,2*j)->Blue = ImageIn(i,j)->Blue;
380               ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
381               ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;
382            }
383
384            if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
385               && j != ImageIn.TellHeight()-1
386               && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
387               && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
388               && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
389               && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
390               && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
391               && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue )
392            {
393                ImageOut(i*2,j*2)->Green =
394                  (ImageIn(i-1,j-1)->Green + ImageIn(i,j-1)->Green +
395                  ImageIn(i-1,j)->Green + 4*ImageIn(i,j)->Green)/8;
396                ImageOut(i*2+1,j*2)->Green =
397                  (ImageIn(i,j-1)->Green + ImageIn(i+1,j-1)->Green +
398                  4*ImageIn(i,j)->Green + ImageIn(i+1,j)->Green)/8;
399                ImageOut(i*2,j*2+1)->Green =
400                  (ImageIn(i-1,j)->Green + 4*ImageIn(i,j)->Green +
401                  ImageIn(i-1,j+1)->Green + ImageIn(i,j+1)->Green)/8;
402                ImageOut(i*2+1,j*2+1)->Green =
403                  (4*ImageIn(i,j)->Green + ImageIn(i+1,j)->Green +
404                  ImageIn(i,j+1)->Green + ImageIn(i+1,j+1)->Green)/8;
405            }
406            else
407            {
408               ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
409               ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
```

```
410             ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
411             ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
412          }
413       }
414     }
415 }
416 
417 // Interpolation with RGB:
418 void rogers3xAlgorithm(BMP ImageIn, BMP &ImageOut)
419 {
420 
421     int NewWidth = (int) ( ImageIn.TellWidth() * 3);
422     int NewHeight = (int) ( ImageIn.TellHeight() * 3);
423 
424     ImageOut.SetSize( NewWidth, NewHeight );
425 
426     if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
427     else ImageOut.SetBitDepth(24);
428 
429     for (int j=0 ; j < ImageIn.TellHeight() ; j++)
430     {
431       for (int i=0 ; i < ImageIn.TellWidth() ; i++)
432       {
433          if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
434             && j != ImageIn.TellHeight()-1
435             && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
436             && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red )
437          {
438             if (ImageIn(i,j-1)->Red == ImageIn(i-1,j)->Red)
                   ImageOut(i*3,j*3)->Red = ImageIn(i,j-1)->Red;
439             else {
440               ImageOut(i*3,j*3)->Red = // E0
441               (ImageIn(i,j-1)->Red +
442               ImageIn(i-1,j)->Red + 2*ImageIn(i,j)->Red)/4;
443             }
444             if (ImageIn(i,j-1)->Red == ImageIn(i+1,j)->Red)
                   ImageOut(i*3+2,j*3)->Red = ImageIn(i,j-1)->Red;
445             else {
446               ImageOut(i*3+2,j*3)->Red = // E2
447               (ImageIn(i,j-1)->Red +
448               2*ImageIn(i,j)->Red + ImageIn(i+1,j)->Red)/4;
449             }
450             if (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red)
                   ImageOut(i*3,j*3+2)->Red = ImageIn(i-1,j)->Red;
451             else {
452               ImageOut(i*3,j*3+2)->Red = // E6
453               (ImageIn(i-1,j)->Red + 2*ImageIn(i,j)->Red +
454                ImageIn(i,j+1)->Red)/4;
455             }
456             if (ImageIn(i+1,j)->Red == ImageIn(i,j+1)->Red)
                   ImageOut(i*3+2,j*3+2)->Red = ImageIn(i+1,j)->Red;
```

```
457                 else {
458                   ImageOut(i*3+2,j*3+2)->Red = // E8
459                   (2*ImageIn(i,j)->Red + ImageIn(i+1,j)->Red +
460                   ImageIn(i,j+1)->Red)/4;
461                 }
462
463                 ImageOut(i*3+1,j*3+1)->Red = ImageIn(i,j)->Red; // E4
464
465                 ImageOut(i*3+1,j*3)->Red = // E1
466                   (2*ImageIn(i,j-1)->Red + 6*ImageIn(i,j)->Red)/8;
467                 ImageOut(i*3,j*3+1)->Red = // E3
468                   (2*ImageIn(i-1,j)->Red + 6*ImageIn(i,j)->Red)/8;
469                 ImageOut(i*3+2,j*3+1)->Red = // E5
470                   (2*ImageIn(i+1,j)->Red + 6*ImageIn(i,j)->Red)/8;
471                 ImageOut(i*3+1,j*3+2)->Red = // E7
472                   (2*ImageIn(i,j+1)->Red + 6*ImageIn(i,j)->Red)/8;
473
474             }
475             else
476             {
477                ImageOut(3*i,3*j)->Red = ImageIn(i,j)->Red;
478                ImageOut(3*i+1,3*j)->Red = ImageIn(i,j)->Red;
479                ImageOut(3*i+2,3*j)->Red = ImageIn(i,j)->Red;
480                ImageOut(3*i,3*j+1)->Red = ImageIn(i,j)->Red;
481                ImageOut(3*i+1,3*j+1)->Red = ImageIn(i,j)->Red;
482                ImageOut(3*i+2,3*j+1)->Red = ImageIn(i,j)->Red;
483                ImageOut(3*i,3*j+2)->Red = ImageIn(i,j)->Red;
484                ImageOut(3*i+1,3*j+2)->Red = ImageIn(i,j)->Red;
485                ImageOut(3*i+2,3*j+2)->Red = ImageIn(i,j)->Red;
486             }
487
488             if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
489                && j != ImageIn.TellHeight()-1
490                && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue
491                && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue )
492             {
493                 if (ImageIn(i,j-1)->Blue == ImageIn(i-1,j)->Blue)
                       ImageOut(i*3,j*3)->Blue = ImageIn(i,j-1)->Blue;
494                 else {
495                   ImageOut(i*3,j*3)->Blue = // E0
496                   (ImageIn(i,j-1)->Blue +
497                   ImageIn(i-1,j)->Blue + 2*ImageIn(i,j)->Blue)/4;
498                 }
499                 if (ImageIn(i,j-1)->Blue == ImageIn(i+1,j)->Blue)
                       ImageOut(i*3+2,j*3)->Blue = ImageIn(i,j-1)->Blue;
500                 else {
501                   ImageOut(i*3+2,j*3)->Blue = // E2
502                   (ImageIn(i,j-1)->Blue +
503                   2*ImageIn(i,j)->Blue + ImageIn(i+1,j)->Blue)/4;
504                 }
505                 if (ImageIn(i-1,j)->Blue == ImageIn(i,j+1)->Blue)
```

```
                ImageOut(i*3,j*3+2)->Blue = ImageIn(i-1,j)->Blue;
506           else {
507             ImageOut(i*3,j*3+2)->Blue = // E6
508             (ImageIn(i-1,j)->Blue + 2*ImageIn(i,j)->Blue +
509             ImageIn(i,j+1)->Blue)/4;
510           }
511           if (ImageIn(i+1,j)->Blue == ImageIn(i,j+1)->Blue)
                ImageOut(i*3+2,j*3+2)->Blue = ImageIn(i+1,j)->Blue;
512           else {
513             ImageOut(i*3+2,j*3+2)->Blue = // E8
514             (2*ImageIn(i,j)->Blue + ImageIn(i+1,j)->Blue +
515             ImageIn(i,j+1)->Blue)/4;
516           }
517
518           ImageOut(i*3+1,j*3+1)->Blue = ImageIn(i,j)->Blue; // E5
519
520           ImageOut(i*3+1,j*3)->Blue = // E1
521             (2*ImageIn(i,j-1)->Blue + 6*ImageIn(i,j)->Blue)/8;
522           ImageOut(i*3,j*3+1)->Blue = // E3
523             (2*ImageIn(i-1,j)->Blue + 6*ImageIn(i,j)->Blue)/8;
524           ImageOut(i*3+2,j*3+1)->Blue = // E5
525             (2*ImageIn(i+1,j)->Blue + 6*ImageIn(i,j)->Blue)/8;
526           ImageOut(i*3+1,j*3+2)->Blue = // E7
527             (2*ImageIn(i,j+1)->Blue + 6*ImageIn(i,j)->Blue)/8;
528
529       }
530       else
531       {
532          ImageOut(3*i,3*j)->Blue = ImageIn(i,j)->Blue;
533          ImageOut(3*i+1,3*j)->Blue = ImageIn(i,j)->Blue;
534          ImageOut(3*i+2,3*j)->Blue = ImageIn(i,j)->Blue;
535          ImageOut(3*i,3*j+1)->Blue = ImageIn(i,j)->Blue;
536          ImageOut(3*i+1,3*j+1)->Blue = ImageIn(i,j)->Blue;
537          ImageOut(3*i+2,3*j+1)->Blue = ImageIn(i,j)->Blue;
538          ImageOut(3*i,3*j+2)->Blue = ImageIn(i,j)->Blue;
539          ImageOut(3*i+1,3*j+2)->Blue = ImageIn(i,j)->Blue;
540          ImageOut(3*i+2,3*j+2)->Blue = ImageIn(i,j)->Blue;
541       }
542
543       if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
544          && j != ImageIn.TellHeight()-1
545          && ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
546          && ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green )
547       {
548           if (ImageIn(i,j-1)->Green == ImageIn(i-1,j)->Green)
                ImageOut(i*3,j*3)->Green = ImageIn(i,j-1)->Green;
549           else {
550             ImageOut(i*3,j*3)->Green = // E0
551             (ImageIn(i,j-1)->Green +
552             ImageIn(i-1,j)->Green + 2*ImageIn(i,j)->Green)/4;
553           }
```

```
554                 if (ImageIn(i,j-1)->Green == ImageIn(i+1,j)->Green)
                        ImageOut(i*3+2,j*3)->Green = ImageIn(i,j-1)->Green;
555                 else {
556                   ImageOut(i*3+2,j*3)->Green = // E2
557                   (ImageIn(i,j-1)->Green +
558                   2*ImageIn(i,j)->Green + ImageIn(i+1,j)->Green)/4;
559                 }
560                 if (ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green)
                        ImageOut(i*3,j*3+2)->Green = ImageIn(i-1,j)->Green;
561                 else {
562                   ImageOut(i*3,j*3+2)->Green = // E6
563                   (ImageIn(i-1,j)->Green + 2*ImageIn(i,j)->Green +
564                   ImageIn(i,j+1)->Green)/4;
565                 }
566                 if (ImageIn(i+1,j)->Green == ImageIn(i,j+1)->Green)
                        ImageOut(i*3+2,j*3+2)->Green = ImageIn(i+1,j)->Green;
567                 else {
568                   ImageOut(i*3+2,j*3+2)->Green = // E8
569                   (2*ImageIn(i,j)->Green + ImageIn(i+1,j)->Green +
570                   ImageIn(i,j+1)->Green)/4;
571                 }
572
573                 ImageOut(i*3+1,j*3+1)->Green = ImageIn(i,j)->Green; // E5
574
575                 ImageOut(i*3+1,j*3)->Green = // E1
576                   (2*ImageIn(i,j-1)->Green + 6*ImageIn(i,j)->Green)/8;
577                 ImageOut(i*3,j*3+1)->Green = // E3
578                   (2*ImageIn(i-1,j)->Green + 6*ImageIn(i,j)->Green)/8;
579                 ImageOut(i*3+2,j*3+1)->Green = // E5
580                   (2*ImageIn(i+1,j)->Green + 6*ImageIn(i,j)->Green)/8;
581                 ImageOut(i*3+1,j*3+2)->Green = // E7
582                   (2*ImageIn(i,j+1)->Green + 6*ImageIn(i,j)->Green)/8;
583
584             }
585             else
586             {
587                ImageOut(3*i,3*j)->Green = ImageIn(i,j)->Green;
588                ImageOut(3*i+1,3*j)->Green = ImageIn(i,j)->Green;
589                ImageOut(3*i+2,3*j)->Green = ImageIn(i,j)->Green;
590                ImageOut(3*i,3*j+1)->Green = ImageIn(i,j)->Green;
591                ImageOut(3*i+1,3*j+1)->Green = ImageIn(i,j)->Green;
592                ImageOut(3*i+2,3*j+1)->Green = ImageIn(i,j)->Green;
593                ImageOut(3*i,3*j+2)->Green = ImageIn(i,j)->Green;
594                ImageOut(3*i+1,3*j+2)->Green = ImageIn(i,j)->Green;
595                ImageOut(3*i+2,3*j+2)->Green = ImageIn(i,j)->Green;
596             }
597          }
598       }
599    }
```

A.3 Rogers2x algorithm

```

#include "EasyBMP_DataStructures.h"
#include "EasyBMP_BMP.h"

struct YUV{
    double Y,U,V;
};
struct RGB{
    double R,G,B;
};

YUV RGB2YUV(RGBApixel In);
RGB YUV2RGB(YUV In);

void rogers2xDownGFX_YUV(BMP ImageIn, BMP &ImageOut)
{
    int NewWidth = (int) (ImageIn.TellWidth() / 2);
    int NewHeight = (int) (ImageIn.TellHeight() / 2);

    ImageOut.SetSize( NewWidth, NewHeight );

    if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
    else ImageOut.SetBitDepth(24);

    for (int j=0 ; j < ImageOut.TellHeight() ; j++)
    {
     for (int i=0 ; i < ImageOut.TellWidth() ; i++)
     {
         if (i != 0 && i != ImageOut.TellWidth()-1 &&
             j != 0 && j != ImageOut.TellHeight()-1)
         {
             YUV p1,p2,p3,p4,pAvg;
             p1 = RGB2YUV(*ImageIn(i*2, j*2));
             p2 = RGB2YUV(*ImageIn(i*2+1,j*2));
             p3 = RGB2YUV(*ImageIn(i*2,j*2+1));
             p4 = RGB2YUV(*ImageIn(i*2+1,j*2+1));

             pAvg.Y = (p1.Y + p2.Y + p3.Y + p4.Y)/4;
             pAvg.U = (p1.U + p2.U + p3.U + p4.U)/4;
             pAvg.V = (p1.V + p2.V + p3.V + p4.V)/4;

             RGB RGBAvg = YUV2RGB(pAvg);

             ImageOut(i,j)->Red = (char) RGBAvg.R;
             ImageOut(i,j)->Green = (char) RGBAvg.G;
             ImageOut(i,j)->Blue = (char) RGBAvg.B;

         }
         else{
```

```
50              ImageOut(i,j)->Red = ImageIn(i*2,j*2)->Red;
51              ImageOut(i,j)->Blue = ImageIn(i*2,j*2)->Blue;
52              ImageOut(i,j)->Green = ImageIn(i*2,j*2)->Green;
53              ImageOut(i,j)->Alpha = ImageIn(i*2,j*2)->Alpha;
54           }
55
56       }
57       if (j % 10 == 0) cout << j << "␣/␣" << ImageOut.TellHeight() << endl;
58      }
59  }
60
61
62  void rogers2xAvgGFX_YUV(BMP ImageIn, BMP &ImageOut)
63  {
64      int NewWidth = (int) (ImageIn.TellWidth() * 2);
65      int NewHeight = (int) (ImageIn.TellHeight() * 2);
66
67      ImageOut.SetSize( NewWidth, NewHeight );
68
69      if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
70      else ImageOut.SetBitDepth(24);
71
72      for (int j=0 ; j < ImageIn.TellHeight() ; j++)
73      {
74       for (int i=0 ; i < ImageIn.TellWidth() ; i++)
75       {
76           if (i != 0 && i != ImageIn.TellWidth()-1 &&
77               j != 0 && j != ImageIn.TellHeight()-1)
78
79           {
80
81               //--------------//
82               // p1 | p2 | p3 //   p5   //-----------//
83               // p4 | p5 | p6 //  --->  // po1 | po2 //
84               // p7 | p8 | p9 //        // po3 | po4 //
85               //--------------//        //-----------//
86
87               YUV p1,p2,p3,p4,p5,p6,p7,p8,p9,po1,po2,po3,po4;
88               p1 = RGB2YUV(*ImageIn(i-1, j-1));
89               p2 = RGB2YUV(*ImageIn(i,j-1));
90               p3 = RGB2YUV(*ImageIn(i+1,j-1));
91               p4 = RGB2YUV(*ImageIn(i-1,j));
92               p5 = RGB2YUV(*ImageIn(i,j));
93               p6 = RGB2YUV(*ImageIn(i+1,j));
94               p7 = RGB2YUV(*ImageIn(i-1,j+1));
95               p8 = RGB2YUV(*ImageIn(i,j+1));
96               p9 = RGB2YUV(*ImageIn(i+1,j+1));
97
98
99               po1.Y = ((p2.Y + p4.Y) + 2*p5.Y)/4;
100              po1.U = ((p2.U + p4.U) + 2*p5.U)/4;
```

```
101             po1.V = ((p2.V + p4.V) + 2*p5.V)/4;
102
103             po2.Y = ((p2.Y + p6.Y) + 2*p5.Y)/4;
104             po2.U = ((p2.U + p6.U) + 2*p5.U)/4;
105             po2.V = ((p2.V + p6.V) + 2*p5.V)/4;
106
107             po3.Y = ((p4.Y + p8.Y) + 2*p5.Y)/4;
108             po3.U = ((p4.U + p8.U) + 2*p5.U)/4;
109             po3.V = ((p4.V + p8.V) + 2*p5.V)/4;
110
111             po4.Y = ((p6.Y + p8.Y) + 2*p5.Y)/4;
112             po4.U = ((p6.U + p8.U) + 2*p5.U)/4;
113             po4.V = ((p6.V + p8.V) + 2*p5.V)/4;
114
115
116             RGB RGBpo1 = YUV2RGB(po1);
117             RGB RGBpo2 = YUV2RGB(po2);
118             RGB RGBpo3 = YUV2RGB(po3);
119             RGB RGBpo4 = YUV2RGB(po4);
120
121             ImageOut(i*2,j*2)->Red = (char) RGBpo1.R;
122             ImageOut(i*2,j*2)->Green = (char) RGBpo1.G;
123             ImageOut(i*2,j*2)->Blue = (char) RGBpo1.B;
124
125             ImageOut(i*2+1,j*2)->Red = (char) RGBpo2.R;
126             ImageOut(i*2+1,j*2)->Green = (char) RGBpo2.G;
127             ImageOut(i*2+1,j*2)->Blue = (char) RGBpo2.B;
128
129             ImageOut(i*2,j*2+1)->Red = (char) RGBpo3.R;
130             ImageOut(i*2,j*2+1)->Green = (char) RGBpo3.G;
131             ImageOut(i*2,j*2+1)->Blue = (char) RGBpo3.B;
132
133             ImageOut(i*2+1,j*2+1)->Red = (char) RGBpo4.R;
134             ImageOut(i*2+1,j*2+1)->Green = (char) RGBpo4.G;
135             ImageOut(i*2+1,j*2+1)->Blue = (char) RGBpo4.B;
136
137         }
138         else{
139            ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;
140            ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;
141            ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;
142            ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;
143
144            ImageOut(2*i,2*j)->Blue = ImageIn(i,j)->Blue;
145            ImageOut(2*i+1,2*j)->Blue = ImageIn(i,j)->Blue;
146            ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
147            ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;
148
149            ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
150            ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
151            ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
```

```
152                 ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
153             }
154 
155         }
156         if (j % 10 == 0) cout << j << "␣/␣" << ImageIn.TellHeight() << endl;
157        }
158 }
159 
160 void rogers2xGFX_YUV(BMP ImageIn, BMP &ImageOut)
161 {
162        int NewWidth = (int) (ImageIn.TellWidth() * 2);
163        int NewHeight = (int) (ImageIn.TellHeight() * 2);
164 
165        ImageOut.SetSize( NewWidth, NewHeight );
166 
167        if (ImageIn.TellBitDepth() == 32) ImageOut.SetBitDepth(32);
168        else ImageOut.SetBitDepth(24);
169 
170        for (int j=0 ; j < ImageIn.TellHeight() ; j++)
171        {
172         for (int i=0 ; i < ImageIn.TellWidth() ; i++)
173         {
174             if (i != 0 && i != ImageIn.TellWidth()-1 &&
175                 j != 0 && j != ImageIn.TellHeight()-1)
176 
177             {
178 
179                 //---------------//
180                 // p1 | p2 | p3 //   p5   //-----------//
181                 // p4 | p5 | p6 //  --->  // po1 | po2 //
182                 // p7 | p8 | p9 //        // po3 | po4 //
183                 //---------------//       //-----------//
184 
185                 YUV p1,p2,p3,p4,p5,p6,p7,p8,p9,po1,po2,po3,po4;
186                 p1 = RGB2YUV(*ImageIn(i-1, j-1));
187                 p2 = RGB2YUV(*ImageIn(i,j-1));
188                 p3 = RGB2YUV(*ImageIn(i+1,j-1));
189                 p4 = RGB2YUV(*ImageIn(i-1,j));
190                 p5 = RGB2YUV(*ImageIn(i,j));
191                 p6 = RGB2YUV(*ImageIn(i+1,j));
192                 p7 = RGB2YUV(*ImageIn(i-1,j+1));
193                 p8 = RGB2YUV(*ImageIn(i,j+1));
194                 p9 = RGB2YUV(*ImageIn(i+1,j+1));
195 
196                 if (ImageIn(i-1,j)->Red == ImageIn(i,j-1)->Red
197                   && ImageIn(i-1,j)->Blue == ImageIn(i,j-1)->Blue
198                   && ImageIn(i-1,j)->Green == ImageIn(i,j-1)->Green)
199                 {
200                     po1.Y = ((p2.Y + p4.Y) + p5.Y)/3;
201                     po1.U = ((p2.U + p4.U) + p5.U)/3;
202                     po1.V = ((p2.V + p4.V) + p5.V)/3;
```

```
203             }
204             else
205             {
206                 po1.Y = ((p2.Y + p4.Y) + 2*p5.Y)/4;
207                 po1.U = ((p2.U + p4.U) + 2*p5.U)/4;
208                 po1.V = ((p2.V + p4.V) + 2*p5.V)/4;
209             }
210
211             if (ImageIn(i,j-1)->Red == ImageIn(i+1,j)->Red
212               && ImageIn(i,j-1)->Blue == ImageIn(i+1,j)->Blue
213               && ImageIn(i,j-1)->Green == ImageIn(i+1,j)->Green)
214             {
215                 po2.Y = ((p2.Y + p6.Y) + p5.Y)/3;
216                 po2.U = ((p2.U + p6.U) + p5.U)/3;
217                 po2.V = ((p2.V + p6.V) + p5.V)/3;
218             }
219             else
220             {
221                 po2.Y = ((p2.Y + p6.Y) + 2*p5.Y)/4;
222                 po2.U = ((p2.U + p6.U) + 2*p5.U)/4;
223                 po2.V = ((p2.V + p6.V) + 2*p5.V)/4;
224             }
225
226             if (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red
227               && ImageIn(i-1,j)->Blue == ImageIn(i,j+1)->Blue
228               && ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green)
229             {
230                 po3.Y = ((p4.Y + p8.Y) + p5.Y)/3;
231                 po3.U = ((p4.U + p8.U) + p5.U)/3;
232                 po3.V = ((p4.V + p8.V) + p5.V)/3;
233             }
234             else
235             {
236                 po3.Y = ((p4.Y + p8.Y) + 2*p5.Y)/4;
237                 po3.U = ((p4.U + p8.U) + 2*p5.U)/4;
238                 po3.V = ((p4.V + p8.V) + 2*p5.V)/4;
239             }
240
241             if (ImageIn(i,j+1)->Red == ImageIn(i+1,j)->Red
242               && ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue
243               && ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green)
244             {
245                 po4.Y = ((p6.Y + p8.Y) + p5.Y)/3;
246                 po4.U = ((p6.U + p8.U) + p5.U)/3;
247                 po4.V = ((p6.V + p8.V) + p5.V)/3;
248             }
249             else
250             {
251                 po4.Y = ((p6.Y + p8.Y) + 2*p5.Y)/4;
252                 po4.U = ((p6.U + p8.U) + 2*p5.U)/4;
253                 po4.V = ((p6.V + p8.V) + 2*p5.V)/4;
```

```
254                 }
255
256                 RGB RGBpo1 = YUV2RGB(po1);
257                 RGB RGBpo2 = YUV2RGB(po2);
258                 RGB RGBpo3 = YUV2RGB(po3);
259                 RGB RGBpo4 = YUV2RGB(po4);
260
261
262                 ImageOut(i*2,j*2)->Red = (char) RGBpo1.R;
263                 ImageOut(i*2,j*2)->Green = (char) RGBpo1.G;
264                 ImageOut(i*2,j*2)->Blue = (char) RGBpo1.B;
265
266                 ImageOut(i*2+1,j*2)->Red = (char) RGBpo2.R;
267                 ImageOut(i*2+1,j*2)->Green = (char) RGBpo2.G;
268                 ImageOut(i*2+1,j*2)->Blue = (char) RGBpo2.B;
269
270                 ImageOut(i*2,j*2+1)->Red = (char) RGBpo3.R;
271                 ImageOut(i*2,j*2+1)->Green = (char) RGBpo3.G;
272                 ImageOut(i*2,j*2+1)->Blue = (char) RGBpo3.B;
273
274                 ImageOut(i*2+1,j*2+1)->Red = (char) RGBpo4.R;
275                 ImageOut(i*2+1,j*2+1)->Green = (char) RGBpo4.G;
276                 ImageOut(i*2+1,j*2+1)->Blue = (char) RGBpo4.B;
277             }
278             else{
279                ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;
280                ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;
281                ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;
282                ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;
283
284                ImageOut(2*i,2*j)->Blue = ImageIn(i,j)->Blue;
285                ImageOut(2*i+1,2*j)->Blue = ImageIn(i,j)->Blue;
286                ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
287                ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;
288
289                ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
290                ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
291                ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
292                ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
293             }
294
295        }
296        if (j % 10 == 0) cout << j << "␣/␣" << ImageIn.TellHeight() << endl;
297      }
298 }
299
300
301
302 YUV RGB2YUV(RGBApixel In)
303 {
304   YUV Out; // legge inn (int), raskere?
```

```
305
306   /* Factors for implementin in HW */
307   Out.Y = In.Red/4 + In.Green/2 + In.Blue/4; // Y
308   Out.U = (In.Blue - Out.Y)/2; //U
309   Out.V = (In.Red - Out.Y)/2; //V
310
311 /* ***Original transform factors***
312   Out.Lum = 0.299*ImageIn(x,y)->Red + 0.587*ImageIn(x,y)->Green +
313     0.114*ImageIn(x,y)->Blue; // Y
314   Out.Krom1 = (ImageIn(x,y)->Blue - Out.Lum)*0.565; //U
315   Out.Krom2 = (ImageIn(x,y)->Red - Out.Lum)*0.713; //V
316 */
317   return Out;
318 }
319
320 RGB YUV2RGB(YUV In)
321 {
322   RGB Out;
323
324   /* Factors for implementin in HW */
325   Out.R = 2*In.V + In.Y;
326   Out.G = In.Y - In.U/2 - In.V/2;
327   Out.B = In.Y + 2*In.U;
328
329 /* ***Original transform faktors***
330   Out.R = 1.403*In.Krom2 + In.Lum;
331   Out.G = In.Lum - 0.344*In.Krom1 - 0.714*In.Krom2;
332   Out.B = In.Lum + 1.77*In.Krom1;
333 */
334   return Out;
335 }
```

B Verilog code

B.1 AXI Read module

```
/****************************************************************************/
/*                                                                          */
/* Author: Roger Skarboe                                                    */
/*                                                                          */
/* Version: v 1.11, 02.06.2010                                              */
/*                                                                          */
/* Description: AMBA AXI interface READ/WRITE. A interface used by ARM.     */
/*              Spesification is given in "AMBA AXI Protocol Specification" */
/*                                                                          */
/****************************************************************************/

module AXI_read(ACLK, ARESETn, ARID, ARADDR, ARLEN, ARSIZE,
  ARBURST, ARLOCK, ARCACHE, ARPROT, ARVALID, ARREADY, RID, RDATA, RRESP,
  RLAST,  RVALID, RREADY, stop_transfer, di, we, start_addr, mem_size_in,
  start_scale);

/* Define constants for output signals */
parameter Id         = 5'b 00000;
parameter Length     = 4'b 0111; // 8 Number of transfer (Table 4-1)
parameter Size       = 3'b 011;  // 8 bytes in transfer (Table 4-2)
parameter Burst_size = 4'b 1000; // 8 bytes in transfer (Table 4-2)
parameter Burst      = 2'b 01;   // INCR (Table 4-3) 2'b 00 for FIXED
parameter Cache      = 4'b 0000; // Noncacheable and nonbufferable (Table 5-1)
parameter Prot       = 3'b 000;  // Normal, secure, data access (Table 5-2)
parameter Lock       = 2'b 00;   // Normal access (Table 6-1)

/* Define constants for input signals */
parameter OKAY = 2'b 00; // encoding of the RRESP[1:0] and BRESP[1:0]
parameter EXOKAY = 2'b 01;
parameter SLVERR = 2'b 10;
parameter DECERR = 2'b 11;
/************************ End of Constant definitions *********************/

/* Global input signals */
input          ACLK;
input          ARESETn;

/* Read address channel signals */
output  [4:0]   ARID;
output [31:0]   ARADDR;
output  [3:0]   ARLEN;
output  [2:0]   ARSIZE;
output  [1:0]   ARBURST;
output  [1:0]   ARLOCK;
output  [3:0]   ARCACHE;
output  [2:0]   ARPROT;
```

```
47 output           ARVALID;
48 input            ARREADY;
49
50 /* Read data channel signals */
51 input   [4:0]    RID;
52 input  [63:0]    RDATA;
53 input   [1:0]    RRESP;
54 input            RLAST;
55 input            RVALID;
56 output           RREADY;
57
58 /* Memory module signals */
59 input            stop_transfer;
60
61 /* RAM signals */
62 output           we;
63 output [63:0]    di;
64
65
66 /* Input from APB */
67 input  [31:0]    start_addr;
68 input  [15:0]    mem_size_in;
69 input            start_scale;
70
71 /****************************************************************************
72 * Register AXI inputs
73 ****************************************************************************/
74 reg              start_scale_r;
75 reg              reset_start_scale_r;
76 reg              first_addr_read;
77
78 /****************************************************************************
79 * AXI Read register space
80 ****************************************************************************/
81 reg   [31:0]     araddr_r;
82 reg              arvalid_r;
83 reg              rready_r;
84 reg              we_r;
85
86
87 //**************************************************************/
88 // Signals to control:
89 // Output: ARVALID, RREADY;
90 //         [31:0] ARADDR;
91 //
92 // Signals to check:
93 // Input:  RLAST, RVALID, ARREADY;
94 //         [1:0]  RRESP;
95 //         [3:0]  RID;
96 //         [31:0] RDATA;
97 //**************************************************************/
```

```
98
99
100
101 /* Process */
102 always @(negedge ARESETn or posedge ACLK)
103 begin
104   if (!ARESETn)
105   begin
106     rready_r  <= 1'b0;
107     we_r      <= 1'b0;
108     arvalid_r <= 1'b0;
109     araddr_r  <= 32'h00000000;
110
111     //start_scale_r <= 1'b0;
112     reset_start_scale_r <= 1'b0;
113     first_addr_read <= 1'b1;
114   end
115
116   // elsif(posedge ACLK)
117   else
118   begin
119
120     // Stop reading in new addresses?
121     if (start_scale_r) // Start scale has been set.
122       arvalid_r <= ~stop_transfer; // stop_transfer is a signal from mem_reg.
123
124
125     // End of frame:
126     if (araddr_r == start_addr + Burst_size*(Length+1)*mem_size_in ) begin
127       arvalid_r <= 1'b0; // Next address not valid
128       reset_start_scale_r <= 1'b1; // Ready for next frame
129       first_addr_read <= 1'b1;
130     end
131     else begin
132       // Increment address:
133       if (ARREADY && arvalid_r) begin // equal sentence insted of counter
134         araddr_r <= araddr_r + Burst_size*(Length+1); // araddr_r += 8*8
135         if (araddr_r == start_addr + Burst_size*(Length+1)*(mem_size_in-1))
136           arvalid_r <= 1'b0;
137       end
138       reset_start_scale_r <= 1'b0;
139     end
140
141
142     // Start reading, first valid address:
143     if (start_scale_r && first_addr_read) // Insure 1 access
144     begin
145       araddr_r <= start_addr; // ADDR from APB
146       arvalid_r <= 1'b1; // Valid address
147       first_addr_read <= 1'b0; // Insure that reset/start happens in 1 clk cycle
148     end
```

```
149
150     end
151   end
152
153
154   always @(posedge start_scale or posedge reset_start_scale_r)
155   begin
156     if (start_scale)
157       start_scale_r <= 1;
158     if (reset_start_scale_r)
159       start_scale_r <= 0;
160   end
161
162
163   /****************************************************************************
164   * Drive outputs
165   ****************************************************************************/
166
167   // Constants:
168   assign ARBURST  = Burst;
169   assign ARLOCK   = Lock;
170   assign ARSIZE   = Size;
171   assign ARPROT   = Prot;
172   assign ARLEN    = Length;
173   assign ARCACHE  = Cache;
174   assign ARID     = Id;
175
176   assign di       = RDATA; // di (RAM) is always RDATA, use WR_EN signal.
177
178   // Registers:
179   assign ARADDR   = araddr_r;
180   assign ARVALID  = arvalid_r;
181   assign RREADY   = 1'b1;
182   assign we       = RVALID;
183
184
185   endmodule
```

B.2 AXI Write module

```
/*****************************************************************************/
/*                                                                         */
/* Author: Roger Skarboe                                                   */
/*                                                                         */
/* Version: v 1.15, 04.06.2010                                             */
/*                                                                         */
/* Description: AMBA AXI interface READ/WRITE. A interface used by ARM.    */
/*              Spesification is given in "AMBA AXI Protocol Specification" */
/*                                                                         */
/*****************************************************************************/


module AXI_write(ACLK, ARESETn, AWID, AWADDR, AWLEN, AWSIZE, AWBURST, AWLOCK,
  AWCACHE,AWPROT, AWVALID, AWREADY, WID, WDATA, WSTRB, WLAST, WVALID,
  WREADY, BID, BRESP, BVALID, BREADY, empty, almost_empty, dout, rd_en,
  data_count, start_scale, dest_addr, mem_size_out, irq_vs, frame_finished);

/* Define constants for output signals */
parameter Id         = 5'b 00000;
parameter Length     = 4'b 0111; // 8 Number of transfer (Table 4-1)
parameter Size       = 3'b 011;  // 8 bytes in transfer (Table 4-2)
parameter Burst_size = 4'b 1000; // 8 bytes in transfer (Table 4-2)
parameter Burst      = 2'b 01;   // INCR (Table 4-3) / FIXED 00?
parameter Cache      = 4'b 0000; // Noncacheable and nonbufferable (Table 5-1)
parameter Prot       = 3'b 000;  // Normal, secure, data access (Table 5-2)
parameter Lock       = 2'b 00;   // Normal access (Table 6-1)
parameter Strb       = 8'h FF;   // 9.2 Write strobes signals

/* Define constants for input signals */
parameter OKAY   = 2'b 00; // encoding of the RRESP[1:0] and BRESP[1:0]
parameter EXOKAY = 2'b 01;
parameter SLVERR = 2'b 10;
parameter DECERR = 2'b 11;
/************************ End of Constant definitions *********************/


/* Global input signals */
input ACLK;
input ARESETn;

/* Write address channel signals */
output [4:0]    AWID;
output [31:0]   AWADDR;
output [3:0]    AWLEN;
output [2:0]    AWSIZE;
output [1:0]    AWBURST;
output [1:0]    AWLOCK;
output [3:0]    AWCACHE;
output [2:0]    AWPROT;
```

```
50 output           AWVALID;
51 input            AWREADY;
52
53 /* Write data channel signals */
54 output [4:0]     WID;
55 output [63:0]    WDATA;
56 output [7:0]     WSTRB;
57 output           WLAST;
58 output           WVALID;
59 input            WREADY;
60
61 /* Write response channel signals */
62 input  [4:0]     BID;
63 input  [1:0]     BRESP;
64 input            BVALID;
65 output           BREADY;
66
67 /* FIFO siganls */
68 input            empty;
69 input            almost_empty;
70 input  [63:0]    dout;
71 input  [9:0]     data_count;
72 output           rd_en;
73
74 /* APB signals */
75 input            start_scale;
76 input  [31:0]    dest_addr;
77 input  [15:0]    mem_size_out;
78 output           irq_vs; // Interrupt: Finished scaling
79
80 /* Input from scale module (Last data written to FIFO) */
81 input            frame_finished;
82
83 /****************************************************************************
84 * AXI Read register space
85 ****************************************************************************/
86
87 /* Register for input signals */
88 reg              start_scale_r;
89 reg              frame_finished_r;
90
91 /* Registers for output signals */
92 reg              awvalid_r;
93 reg              wlast_r;
94 reg              wvalid_r;
95 reg              rd_en_r;
96 reg    [31:0]    awaddr_r;
97 reg              addr_finished_r;
98
99 /* Variables */
100 reg   [2:0]     countToLast; // count to Length (Length=8) 7 down to 0.
```

```
101 reg             burst_in_FIFO;
102 reg             first_FIFO_read; // get rid of the first FIFO data...
103 reg             first_addr_write;
104 reg             reset_start_scale_r;
105
106 //**********************************************************/
107 // Signals to control:
108 // Output: AWVALID, WLAST, WVALID, BREADY;
109 //         [31:0] AWADDR, WDATA;
110 //
111 // Signals to check:
112 // Input:  AWREADY, WREADY, BVALID;
113 //         [1:0]  BRESP;
114 //         [3:0]  BID;
115 //**********************************************************/
116
117
118 /* Process */
119 always @(negedge ARESETn or posedge ACLK)
120 begin
121   if (!ARESETn)
122   begin
123     awvalid_r        <= 1'b0;
124     awaddr_r         <= 32'h00010000;
125     wlast_r          <= 1'b0;
126     wvalid_r         <= 1'b0;
127     rd_en_r          <= 1'b0;
128     addr_finished_r  <= 1'b0;
129
130     frame_finished_r <= 1'b0;
131     //start_scale_r    <= 1'b0;
132     reset_start_scale_r <= 1'b0;
133
134     countToLast      <= Length; // Length-1, count down to 0.
135     burst_in_FIFO    <= 1'b0;
136     first_FIFO_read  <= 1'b1;
137     first_addr_write <= 1'b1;
138   end
139
140   else // elsif(posedge ACLK)
141   begin
142
143     reset_start_scale_r <= 0;
144
145     // Increment address:
146     if (AWREADY && awvalid_r)
147       awaddr_r <= awaddr_r + Burst_size*(Length+1); // awaddr_r += 8*8
148     // Finished writing address (AWVALID <= 0):
149     if (awaddr_r == ( dest_addr + Burst_size*(Length+1)*mem_size_out ) ) begin
150       awvalid_r <= 1'b0;
151       addr_finished_r <= 1'b1;
```

```
152     end
153
154     // Read data from FIFO: (Moved up now...)
155     wlast_r <= 1'b0; // Default 0: Not last byte read.
156     rd_en_r <= 1'b0; //!! Default 0: Not enable read on FIFO.
157
158     if (countToLast == 0)
159       wlast_r <= 1;
160
161     // Read number of burst:
162     if (WREADY && wvalid_r) begin // wvalid_r
163       rd_en_r <= 1'b1; // Tell FIFO that data is read
164       if (countToLast == 0) begin
165         wlast_r <= 0;
166         burst_in_FIFO <= 0;
167         wvalid_r <= 0;
168         countToLast <= Length;
169         rd_en_r <= 0;
170       end
171       else
172         if (countToLast == 1)
173           wlast_r <= 1'b1;
174         countToLast <= countToLast - 1;
175     end
176
177
178     // If FIFO holds at least a burst, write on AXI Write bus:
179     if (data_count > Burst_size-1) begin
180       burst_in_FIFO <= 1;
181       if (first_FIFO_read) begin // FIFO holds reset value first
182         wvalid_r <= 0;
183         first_FIFO_read <= 0;
184       end
185       else
186         wvalid_r <= 1;
187
188       if (WREADY) // If AXI master is ready, set read enable on FIFO
189         rd_en_r <= 1;
190     end
191
192
193     // Signal from scale module that last data packet is written to FIFO.
194     if (frame_finished) begin
195       frame_finished_r <= 1;
196       reset_start_scale_r <= 1;
197     end
198
199
200
201     // Start reading to destination, first valid address:
202     if (start_scale_r && first_addr_write) begin // Insure 1 access
```

```
203       awaddr_r         <= dest_addr;
204       awvalid_r        <= 1'b1;
205       frame_finished_r <= 1'b0;
206       addr_finished_r  <= 1'b0;
207       countToLast      <= Length;
208       //start_scale_r    <= 1'b1;
209       burst_in_FIFO    <= 1'b0;
210       //first_FIFO_read  <= 1'b1;
211       first_addr_write <= 1'b0;
212     end
213
214   end
215 end
216
217 always @(posedge start_scale or posedge reset_start_scale_r)
218 begin
219   if (start_scale)
220     start_scale_r <= 1;
221   if (reset_start_scale_r)
222     start_scale_r <= 0;
223 end
224
225
226 /****************************************************************************
227 * Drive outputs
228 ****************************************************************************/
229
230 // Assign output constants:
231 assign AWBURST  = Burst;
232 assign AWLOCK   = Lock;
233 assign AWSIZE   = Size;
234 assign AWPROT   = Prot;
235 assign AWLEN    = Length;
236 assign AWCACHE  = Cache;
237 assign AWID     = Id;
238 assign WID      = Id;
239 assign WSTRB    = Strb;
240
241 assign WDATA    = dout; // WDATA is wired to output of FIFO.
242 assign BREADY   = 1'b1; // No use of response channel
243
244 // Registers:
245 assign AWVALID  = awvalid_r;
246 assign AWADDR   = awaddr_r;
247 assign WLAST    = wlast_r;
248 assign WVALID   = wvalid_r;
249 assign rd_en    = WREADY && burst_in_FIFO;//rd_en_r;
250
251 // IRQ when: FIFO is empty, finished writing addresses, scale module finished.
252 assign irq_vs   = (almost_empty || empty) && addr_finished_r && frame_finished_r;
253
```

```
254 endmodule
```

B.3 APB module

```verilog
/****************************************************************************/
/*                                                                          */
/* Author: Roger Skarboe                                                    */
/*                                                                          */
/* Version: v 1.04, 12.05.2010                                              */
/*                                                                          */
/* Description: APB slave for Video scaler                                  */
/*                                                                          */
/****************************************************************************/

module apb_slv
  (PCLK, PRESETn,
  PADDR, PSEL, PENABLE, PWRITE, PWDATA, PREADY, PRDATA, PSLVERR,
  start_addr, dest_addr, mem_size_in, mem_size_out, start_scale,
  line_width_in, frame_width_out, frame_height_out, irq_vs);

// Clock and Reset
input           PCLK;
input           PRESETn;
// APB Slave interface
input  [31:0]   PADDR;
input           PSEL;
input           PENABLE;
input           PWRITE;
input  [31:0]   PWDATA;
output [31:0]   PRDATA;
output          PREADY;
output          PSLVERR;
// Signals connections to other modules:
output [31:0]   start_addr;
output [31:0]   dest_addr;
output [15:0]   mem_size_in;
output [15:0]   mem_size_out;
output          start_scale;
output [10:0]   line_width_in; // max 1920
output [10:0]   frame_width_out; // max 1920
output [10:0]   frame_height_out; // max 1080
input           irq_vs;

/*****************************************************************************
* Register APB inputs
*****************************************************************************/
reg             pwrite_r;
reg  [31:0]     pwdata_r;
reg  [31:0]     paddr_r;

wire            apb_en = (PSEL || pwrite_r);

always @(posedge PCLK) begin
```

```
50    if (apb_en) begin
51      paddr_r <= PADDR;
52      pwdata_r <= PWDATA;
53    end
54  end
55
56  always @(posedge PCLK or negedge PRESETn) begin
57    if (!PRESETn)
58      pwrite_r <= 1'b0;
59    else  // if(apb_en) pwrite_r <= PSEL && PWRITE && !PENABLE else the same
60      pwrite_r <= ( apb_en ? PSEL && PWRITE && !PENABLE : pwrite_r );
61  end
62
63  /******************************************************************************
64  * APB register space and interrupt handling.
65  ******************************************************************************/
66  `define APB_START_ADDR    12'h00 // Physical start address
67  `define APB_DEST_ADDR     12'h04 // Physical destination address
68  `define APB_MEM_IN_SIZE   12'h08 // Set size to read
69  `define APB_MEM_OUT_SIZE  12'h0c // Set size to write
70  `define APB_START_SCALE   12'h10 // To start scaling
71  `define APB_FRAME_W_IN    12'h14 // Set input frame width
72  `define APB_FRAME_W_OUT   12'h18 // Set output frame width
73  `define APB_FRAME_H_OUT   12'h1c // Set output frame heigth
74  `define APB_TOTAL_COUNT   12'h20 // Read total count
75
76  reg  [31:0]     start_addr_r;
77  reg  [31:0]     dest_addr_r;
78  reg  [15:0]     mem_size_in_r;
79  reg  [15:0]     mem_size_out_r;
80  reg             start_scale_r;
81  reg  [10:0]     line_width_in_r;
82  reg  [10:0]     frame_width_out_r;
83  reg  [10:0]     frame_height_out_r;
84  reg  [31:0]     read_data;
85  reg  [23:0]     total_clk_count;
86  reg             en_count;
87  reg             irq_vs_r;
88
89  wire [11:0]     cfg_addr = paddr_r[11:0];
90
91
92  // APB read
93  always @(paddr_r or start_addr_r or dest_addr_r or mem_size_in_r or
94    mem_size_out_r or start_scale_r or line_width_in_r or frame_width_out_r or
95    frame_height_out_r or total_clk_count or cfg_addr) begin
96    case (cfg_addr)
97      `APB_START_ADDR:    read_data = start_addr_r;
98      `APB_DEST_ADDR:     read_data = dest_addr_r;
99      `APB_MEM_IN_SIZE:   read_data = { 16'b0, mem_size_in_r };
100     `APB_MEM_OUT_SIZE:  read_data = { 16'b0, mem_size_out_r };
```

```
101     `APB_START_SCALE:   read_data = { 31'b0, start_scale_r };
102     `APB_FRAME_W_IN:    read_data = { 21'b0, line_width_in_r };
103     `APB_FRAME_W_OUT:   read_data = { 21'b0, frame_width_out_r };
104     `APB_FRAME_H_OUT:   read_data = { 21'b0, frame_height_out_r };
105     `APB_TOTAL_COUNT:   read_data = {  8'b0, total_clk_count };
106     default:            read_data = 32'h00000000;
107   endcase
108 end
109
110
111 always @(posedge PCLK or negedge PRESETn) begin
112   if (!PRESETn) begin
113     start_addr_r <= 32'hffffffff;
114     dest_addr_r <= 32'hffffffff;
115     mem_size_in_r <= 16'h0000;
116     mem_size_out_r <= 16'h0000;
117     start_scale_r <= 1'b0;
118     line_width_in_r <= 11'h000;
119     frame_width_out_r <= 11'h000;
120     frame_height_out_r <= 11'h000;
121
122     total_clk_count <= 24'h000000;
123     //en_count <= 1'b0;
124   end
125   else begin
126     if (pwrite_r && cfg_addr == `APB_START_ADDR)
127       start_addr_r <= pwdata_r;
128
129     if (pwrite_r && cfg_addr == `APB_DEST_ADDR)
130       dest_addr_r <= pwdata_r;
131
132     if (pwrite_r && cfg_addr == `APB_START_SCALE)
133       start_scale_r <= pwdata_r[0];
134
135     if (pwrite_r && cfg_addr == `APB_MEM_IN_SIZE)
136       mem_size_in_r <= pwdata_r[15:0];
137
138     if (pwrite_r && cfg_addr == `APB_MEM_OUT_SIZE)
139       mem_size_out_r <= pwdata_r[15:0];
140
141     if (pwrite_r && cfg_addr == `APB_FRAME_W_IN)
142       line_width_in_r <= pwdata_r[10:0];
143
144     if (pwrite_r && cfg_addr == `APB_FRAME_W_OUT)
145       frame_width_out_r <= pwdata_r[10:0];
146
147     if (pwrite_r && cfg_addr == `APB_FRAME_H_OUT)
148       frame_height_out_r <= pwdata_r[10:0];
149
150     irq_vs_r <= irq_vs;
151
```

```
152
153       // Count total clock periods used to scale:
154       if (en_count)
155         total_clk_count <= total_clk_count + 1;
156
157     end
158   end
159
160   // Total clk count process:
161   always @(posedge start_scale_r or posedge irq_vs_r)
162   begin
163     if (start_scale_r)
164       en_count <= 1'b1;
165     if (irq_vs_r)
166       en_count <= 1'b0;
167   end
168
169
170   /*******************************************************************************
171   * Drive outputs
172   *******************************************************************************/
173
174   assign PRDATA = read_data;
175   assign PREADY = 1'b1;
176   assign PSLVERR = 1'b0;
177   assign start_addr = start_addr_r;
178   assign dest_addr = dest_addr_r;
179   assign mem_size_in = mem_size_in_r;
180   assign mem_size_out = mem_size_out_r;
181   assign start_scale = start_scale_r;
182   assign line_width_in = line_width_in_r;
183   assign frame_width_out = frame_width_out_r;
184   assign frame_height_out = frame_height_out_r;
185
186   endmodule
```

B.4 Memory Control module (mem_reg)

```
/****************************************************************************/
/*                                                                          */
/* Author: Roger Skarboe                                                    */
/*                                                                          */
/* Version: v 1.04, 05.06.2010, for scale1875                               */
/*                                                                          */
/* Description: Module to register scan lines from AXI Read module.         */
/*              Module gets input from AXI Read module.                     */
/*              Module gets input from cooperation scale module.            */
/*              The module should keep track of sufficient number of scan   */
/*              lines. (Minimum of 2 lines)                                 */
/*                                                                          */
/****************************************************************************/

module mem_reg(ACLK, ARESETn, write_enable, stop_transfer, ram_data,
  aw, ar, reg_shift, reg_ready, line1, line2, line_width_in, FIFO_full);

/******************* Define constants for output signals *******************/
parameter BUS_WIDTH = 64;
parameter ADDR_WIDTH = 10;
//parameter RAM_DEPTH = 1024;
//parameter LINE_WIDTH = 16; // 128 / 8(bytes) = 16

/*********************** End of Constant definitions **********************/

/* Global input signals */
input                     ACLK;
input                     ARESETn;

/* AXI Read signals */
input                     write_enable;
output                    stop_transfer;

/* RAM signals */
input   [BUS_WIDTH-1:0]   ram_data;
output  [ADDR_WIDTH-1:0]  aw; // Write address
output  [ADDR_WIDTH-1:0]  ar; // Read address

/* Register signals - Scale module signals */
input                     reg_shift; // Oppdate registers
input                     FIFO_full;
output                    reg_ready; // Tell if registers has data ready
output  [255:0]           line1;
output  [255:0]           line2;

/* APB signals */
input   [10:0]            line_width_in;

/****************************************************************************
```

```
50  * Register space
51  ****************************************************************************/
52  reg                           stop_transfer_r;
53  reg      [ADDR_WIDTH:0]       ar_r; // one extra bit to insure that aw_r > ar_r
54  reg      [ADDR_WIDTH:0]       aw_r; // one extra bit to insure that aw_r > ar_r
55  reg                           reg_ready_r;
56  wire     [7:0]                LINE_WIDTH;
57
58  // Counters:
59  reg      [7:0]                count_line_width; // max: 1920/8 = 240 (h'F0)
60  reg      [2:0]                count_first_four;
61  reg      [1:0]                reg_pos;
62  reg      [3:0]                line_number;
63  reg      [15:0]               dummy_mem_count;
64
65  // Variabel:
66  reg                           on_line1;
67
68  // Wire l1 and l2 to line1 and line2 registers:
69  reg     [63:0]               l1 [0:3];
70  reg     [63:0]               l2 [0:3];
71
72  /************************** End of regsiter space *************************/
73
74
75  /* Assign constant */
76  assign LINE_WIDTH = line_width_in [10:3]; // Set from APB. line width / 8.
77
78
79  /* Process */
80  always @(negedge ARESETn or posedge ACLK)
81  begin
82    if (!ARESETn)
83    begin
84      // Output registers:
85      stop_transfer_r <= 1'b0;
86      aw_r <= 0;
87      ar_r <= 0;
88      reg_ready_r <= 1'b0;
89      //line1_r <= 0;
90      //line2_r <= 0;
91      l1[0] <= 0; l1[1] <= 0; l1[2] <= 0; l1[3] <= 0;
92      l2[0] <= 0; l2[1] <= 0; l2[2] <= 0; l2[3] <= 0;
93      // Counters:
94      count_line_width <= 0;
95      count_first_four <= 0;
96      reg_pos <= 0;
97      line_number <= 0;
98      dummy_mem_count <= 0;
99      // Varable:
100     on_line1 <= 1;
```

```
101     end
102
103     // elsif(posedge ACLK)
104     else
105     begin
106
107       // Update address on RAM when a write_enable accurs:
108       if (write_enable) begin // same as we on RAM
109           aw_r <= aw_r + 1;
110       end
111
112
113       // Initial start: (first line?)  // Latency of 2 when updating ar (address
            read)
114       if (count_first_four < 4) begin // != 4?
115         reg_ready_r <= 0;
116         if (aw_r != ar_r && aw_r-1 != ar_r) begin
117           l1[count_first_four] <= ram_data;
118           ar_r <= ar_r + 1;
119           count_first_four <= count_first_four + 1; // Reset?
120           reg_ready_r <= 1;
121           count_line_width <= 4;
122           on_line1 <= 1;
123         end
124       end
125
126
127       // Normal sequense:
128       else if (aw_r != ar_r && reg_shift && !FIFO_full) begin // change reg_shift
            in scale125!
129
130         // Read part of Line1 from RAM
131         if (on_line1) begin // reg_ready_r
132           on_line1 <= 0;
133           reg_ready_r <= 0;
134           l1[reg_pos] <= ram_data;
135           ar_r <= ar_r + LINE_WIDTH; // Read second line next
136         end
137
138         // Read part of Line2 from RAM
139         else begin
140           on_line1 <= 1;
141           reg_ready_r <= 1;
142           l2[reg_pos] <= ram_data;
143           count_line_width <= count_line_width + 1;
144           dummy_mem_count <= dummy_mem_count + 1; // For testing..
145
146           if (count_line_width == LINE_WIDTH-1) begin
147
148             count_line_width <= 0;
149             line_number <= line_number + 1;
```

```
150
151           // test case sentence instead...
152
153           if (line_number == 0 || line_number == 2 ||
154           line_number == 4 || line_number == 6 ||
155           line_number == 8 || line_number == 10 ||
156           line_number == 12 || line_number == 13) begin
157             ar_r <= ar_r - LINE_WIDTH - LINE_WIDTH + 1;
158           end
159
160           else if (line_number == 14) begin
161             ar_r <= ar_r + 1;
162             line_number <= 0;
163           end
164           else
165             ar_r <= ar_r - LINE_WIDTH + 1;
166
167         end
168
169         else
170           ar_r <= ar_r - LINE_WIDTH + 1; // Read first line next
171
172
173         if (reg_pos == 3) // reg_pos: 0->1->2->3->0->1..
174           reg_pos <= 0;
175         else
176           reg_pos <= reg_pos + 1;
177       end
178
179     end
180
181     else if (aw_r == ar_r || FIFO_full)
182       reg_ready_r <= 0;
183
184     else // reg_shift = 0, calculate new without changing registers.
185       reg_ready_r <= 1;
186
187
188     // Controll of RAM (ar must be behind aw):
189     if (on_line1) begin
190       if (aw_r > ar_r + 256) // random: find minimum! (256 addr ahead....)
191         stop_transfer_r <= 1; // Stop reading in new addresses no AXI read
192       else
193         stop_transfer_r <= 0;
194     end
195     if (ar_r[ADDR_WIDTH-1:0] == 0) begin
196       aw_r[ADDR_WIDTH] <= 0;
197       ar_r[ADDR_WIDTH] <= 0;
198     end
199
200   end // elsif(posedge ACLK)
```

```
201 end // always
202
203
204 /*****************************************************************************
205 * Drive outputs
206 *****************************************************************************/
207 assign stop_transfer = stop_transfer_r;
208 assign aw = aw_r [ADDR_WIDTH-1:0];
209 assign ar = ar_r [ADDR_WIDTH-1:0];
210 assign reg_ready = reg_ready_r;
211 assign line1 = {l1[0], l1[1], l1[2], l1[3]};
212 assign line2 = {l2[0], l2[1], l2[2], l2[3]};
213
214 endmodule
```

B.5 Scale 1.25 module

```
/*************************************************************************/
/*                                                                       */
/* Author: Roger Skarb                                                   */
/*                                                                       */
/* Version: v 1.05, 29.05.2010                                           */
/*                                                                       */
/* Description: Module to scale from SD to HD (scale factor: 1.25).      */
/*              Module gets input from AXI Read module.                  */
/*              Module gets input from cooperation register module.      */
/*                                                                       */
/*************************************************************************/

module scale125(ACLK, ARESETn, din, FIFO_full, wr_en, frame_finished,
  reg_ready, reg_shift, line1_r, line2_r, frame_width_o, frame_height_o);

/* Constants needed for task/function */
parameter a00 = 5'b 00000;
parameter a01 = 5'b 00001;
parameter a02 = 5'b 00010;
parameter a03 = 5'b 00011;
parameter a04 = 5'b 00100;
parameter a06 = 5'b 00110;
parameter a08 = 5'b 01000;
parameter a09 = 5'b 01001;
parameter a12 = 5'b 01100;
parameter a16 = 5'b 10000;

/*********************** End of Constant definitions ********************/

/* Global input signals */
input           ACLK;
input           ARESETn;

/* FIFO output signals */
output [63:0]   din; // Assign to data in channel on FIFO
output          wr_en;
input           FIFO_full; // Assign FIFOs almost full


/* AXI Write signals */
output          frame_finished; // Used for IRQ on AXI Write

/* Register signals */
input           reg_ready; // Tell if registers har data ready
output          reg_shift; // Oppdate registers

/* APB signals */
input  [10:0]   frame_width_o;
input  [10:0]   frame_height_o;
```

```
50
51 /*****************************************************************************
52 * Register space
53 *****************************************************************************/
54 input [255:0]   line1_r, line2_r;
55 reg             wr_en_r;
56 reg             frame_finished_r;
57 reg             reg_shift_r;
58 reg   [63:0]    fifo_data;
59 wire  [7:0]     FRAME_WIDTH_OUT;
60 wire  [10:0]    FRAME_HEIGHT_OUT;
61
62
63 // Counters:
64 reg   [2:0]     line_number; // 5 line number used. Toggle between 5 lines.
65 reg   [2:0]     line_count;  // 5 different ways to calculate output per line.
66 reg   [8:0]     horizontal_count; // +1 for every 5 output pixels. Max: 256
67 reg   [7:0]     vertical_count; // +1 for every 5 output lines in frame. (144)
68 reg   [1:0]     yuv_count;
69
70 reg   [15:0]    dummy_count;
71 reg   [15:0]    dummy_FIFO_full_count;
72 reg             last_data_packet; // Ensure that last data is read.
73
74
75 // Input register for task (avg_func):
76 reg [4:0] s1 [0:7];
77 reg [4:0] s2 [0:7];
78 reg [4:0] s3 [0:7];
79 reg [4:0] s4 [0:7];
80 reg [7:0] pix1 [0:7];
81 reg [7:0] pix2 [0:7];
82 reg [7:0] pix3 [0:7];
83 reg [7:0] pix4 [0:7];
84
85 // Wire p1 and p2 to line1 and line2 register:
86 wire [7:0] p1 [0:31];
87 wire [7:0] p2 [0:31];
88
89
90 assign {p1[7], p1[6], p1[5], p1[4], p1[3], p1[2], p1[1], p1[0], p1[15], p1[14],
91 p1[13], p1[12], p1[11], p1[10], p1[9], p1[8], p1[23], p1[22], p1[21], p1[20],
92 p1[19], p1[18], p1[17], p1[16], p1[31], p1[30], p1[29], p1[28], p1[27], p1[26],
93 p1[25], p1[24]} = line1_r;
94
95 assign {p2[7], p2[6], p2[5], p2[4], p2[3], p2[2], p2[1], p2[0], p2[15], p2[14],
96 p2[13], p2[12], p2[11], p2[10], p2[9], p2[8], p2[23], p2[22], p2[21], p2[20],
97 p2[19], p2[18], p2[17], p2[16], p2[31], p2[30], p2[29], p2[28], p2[27], p2[26],
98 p2[25], p2[24]} = line2_r;
99
100
```

```
101
102 /* Assign constants */
103 assign FRAME_WIDTH_OUT  = frame_width_o [10:3];
104 assign FRAME_HEIGHT_OUT = frame_height_o;
105
106
107 /***************************************************************************
108 * Function / Task declaration
109 ***************************************************************************/
110
111 task avg_func;
112   input [4:0] s1_r,s2_r,s3_r,s4_r; // s1+s2+s3+s4 = 16 // 4:0
113   input [7:0] pix1_r,pix2_r,pix3_r,pix4_r; // pixel value
114   output [7:0] po;
115   reg [11:0] temp;
116   begin
117
118     if      (s1_r[4] == 1'b1) po = pix1_r;
119     else if (s2_r[4] == 1'b1) po = pix2_r;
120     else if (s3_r[4] == 1'b1) po = pix3_r;
121     else if (s4_r[4] == 1'b1) po = pix4_r;
122     else begin
123       temp = ((pix1_r*s1_r[3:0]) + (pix2_r*s2_r[3:0]) + (pix3_r*s3_r[3:0]) +
            (pix4_r*s4_r[3:0]));
124       po = temp[11:4]; // divide by 16
125     end
126   end
127 endtask
128
129 /* Process */
130 always @(pix1[0] or pix1[1] or pix1[2] or pix1[3] or pix1[4] or pix1[5] or
131 pix1[6] or pix1[7] or pix2[0] or pix2[1] or pix2[2] or pix2[3] or pix2[4] or
132 pix2[5] or pix2[6] or pix2[7] or pix3[0] or pix3[1] or pix3[2] or pix3[3] or
133 pix3[4] or pix3[5] or pix3[6] or pix3[7] or pix4[0] or pix4[1] or pix4[2] or
134 pix4[3] or pix4[4] or pix4[5] or pix4[6] or pix4[7] or
135 s1[0] or s1[1] or s1[2] or s1[3] or s1[4] or s1[5] or s1[6] or s1[7] or
136 s2[0] or s2[1] or s2[2] or s2[3] or s2[4] or s2[5] or s2[6] or s2[7] or
137 s3[0] or s3[1] or s3[2] or s3[3] or s3[4] or s3[5] or s3[6] or s3[7] or
138 s4[0] or s4[1] or s4[2] or s4[3] or s4[4] or s4[5] or s4[6] or s4[7]) begin
139
140  avg_func(s1[0],s2[0],s3[0],s4[0], pix1[0],pix2[0],pix3[0],pix4[0],
       fifo_data[7:0]);
141  avg_func(s1[1],s2[1],s3[1],s4[1], pix1[1],pix2[1],pix3[1],pix4[1],
       fifo_data[15:8]);
142  avg_func(s1[2],s2[2],s3[2],s4[2], pix1[2],pix2[2],pix3[2],pix4[2],
       fifo_data[23:16]);
143  avg_func(s1[3],s2[3],s3[3],s4[3], pix1[3],pix2[3],pix3[3],pix4[3],
       fifo_data[31:24]);
144  avg_func(s1[4],s2[4],s3[4],s4[4], pix1[4],pix2[4],pix3[4],pix4[4],
       fifo_data[39:32]);
145  avg_func(s1[5],s2[5],s3[5],s4[5], pix1[5],pix2[5],pix3[5],pix4[5],
```

```
        fifo_data[47:40]);
146  avg_func(s1[6],s2[6],s3[6],s4[6], pix1[6],pix2[6],pix3[6],pix4[6],
        fifo_data[55:48]);
147  avg_func(s1[7],s2[7],s3[7],s4[7], pix1[7],pix2[7],pix3[7],pix4[7],
        fifo_data[63:56]);
148
149 end
150
151
152
153 /* Process */
154 always @(negedge ARESETn or posedge ACLK)
155 begin
156   if (!ARESETn)
157   begin
158     // Output registers:
159     reg_shift_r <= 1'b0;
160     frame_finished_r <= 1'b0;
161     wr_en_r <= 1'b0;
162     // Counters:
163     horizontal_count <= 0;
164     vertical_count <= 0;
165     line_number <= 3'b000;
166     line_count <= 3'b000;
167
168     last_data_packet <= 0;
169     dummy_count <= 0;
170     dummy_FIFO_full_count <= 0;
171     yuv_count <= 0;
172     // Reset intern registers? (+ 123 LUTS (20 % more), + 14MHz (4,3%))
173     //s1 <= 0; s2 <= 0; s3 <= 0; s4 <= 0;
174     //pix1 <= 0; pix2 <= 0; pix3 <= 0; pix4 <= 0;
175   end
176
177   // else if(posedge ACLK)
178   else begin
179
180     frame_finished_r <= 1'b0; // Default value
181     if (FIFO_full) // If FIFO is full hold last reg_shift value
182       reg_shift_r <= reg_shift_r;
183     else
184       reg_shift_r <= 1'b1; // Default value
185
186     wr_en_r <= 1'b0; // Defaukt value
187
188     // Start
189     if (reg_ready || last_data_packet)
190     begin
191
192       wr_en_r <= 1'b1;
193       last_data_packet <= 0;
```

```
194
195       if (line_count == 3'b000) begin
196
197         pix1[0] <= p1[0]; pix2[0] <= p1[1]; pix3[0] <= p2[0]; pix4[0] <= p2[1];
198         pix1[1] <= p1[0]; pix2[1] <= p1[1]; pix3[1] <= p2[0]; pix4[1] <= p2[1];
199         pix1[2] <= p1[1]; pix2[2] <= p1[2]; pix3[2] <= p2[1]; pix4[2] <= p2[2];
200         pix1[3] <= p1[2]; pix2[3] <= p1[3]; pix3[3] <= p2[2]; pix4[3] <= p2[3];
201         pix1[4] <= p1[2]; pix2[4] <= p1[3]; pix3[4] <= p2[2]; pix4[4] <= p2[3];
202         pix1[5] <= p1[4]; pix2[5] <= p1[5]; pix3[5] <= p2[4]; pix4[5] <= p2[5];
203         pix1[6] <= p1[4]; pix2[6] <= p1[5]; pix3[6] <= p2[4]; pix4[6] <= p2[5];
204         pix1[7] <= p1[5]; pix2[7] <= p1[6]; pix3[7] <= p2[5]; pix4[7] <= p2[6];
205
206         line_count <= line_count + 1;
207
208         reg_shift_r <= 1'b0;
209       end
210
211       else if (line_count == 3'b001) begin
212
213         pix1[0] <= p1[6];  pix2[0] <= p1[7];  pix3[0] <= p2[6];  pix4[0] <=
               p2[7];
214         pix1[1] <= p1[6];  pix2[1] <= p1[7];  pix3[1] <= p2[6];  pix4[1] <=
               p2[7];
215         pix1[2] <= p1[8];  pix2[2] <= p1[9];  pix3[2] <= p2[8];  pix4[2] <=
               p2[9];
216         pix1[3] <= p1[8];  pix2[3] <= p1[9];  pix3[3] <= p2[8];  pix4[3] <=
               p2[9];
217         pix1[4] <= p1[9];  pix2[4] <= p1[10]; pix3[4] <= p2[9];  pix4[4] <=
               p2[10];
218         pix1[5] <= p1[10]; pix2[5] <= p1[11]; pix3[5] <= p2[10]; pix4[5] <=
               p2[11];
219         pix1[6] <= p1[10]; pix2[6] <= p1[11]; pix3[6] <= p2[10]; pix4[6] <=
               p2[11];
220         pix1[7] <= p1[12]; pix2[7] <= p1[13]; pix3[7] <= p2[12]; pix4[7] <=
               p2[13];
221
222         line_count <= line_count + 1;
223
224         // SHIFT IN NEW DATA IN [255:192]
225         reg_shift_r <= 1'b1;
226       end
227
228       else if (line_count == 3'b010) begin
229
230         pix1[0] <= p1[12]; pix2[0] <= p1[13]; pix3[0] <= p2[12]; pix4[0] <=
               p2[13];
231         pix1[1] <= p1[13]; pix2[1] <= p1[14]; pix3[1] <= p2[13]; pix4[1] <=
               p2[14];
232         pix1[2] <= p1[14]; pix2[2] <= p1[15]; pix3[2] <= p2[14]; pix4[2] <=
               p2[15];
233         pix1[3] <= p1[14]; pix2[3] <= p1[15]; pix3[3] <= p2[14]; pix4[3] <=
```

```
        p2[15];
234       pix1[4] <= p1[16]; pix2[4] <= p1[17]; pix3[4] <= p2[16]; pix4[4] <=
             p2[17];
235       pix1[5] <= p1[16]; pix2[5] <= p1[17]; pix3[5] <= p2[16]; pix4[5] <=
             p2[17];
236       pix1[6] <= p1[17]; pix2[6] <= p1[18]; pix3[6] <= p2[17]; pix4[6] <=
             p2[18];
237       pix1[7] <= p1[18]; pix2[7] <= p1[19]; pix3[7] <= p2[18]; pix4[7] <=
             p2[19];
238
239       line_count <= line_count + 1;
240
241       // SHIFT IN NEW DATA IN [191:128]
242       reg_shift_r <= 1'b1;
243     end
244
245     else if (line_count == 3'b011) begin
246
247       pix1[0] <= p1[18]; pix2[0] <= p1[19]; pix3[0] <= p2[18]; pix4[0] <=
             p2[19];
248       pix1[1] <= p1[20]; pix2[1] <= p1[21]; pix3[1] <= p2[20]; pix4[1] <=
             p2[21];
249       pix1[2] <= p1[20]; pix2[2] <= p1[21]; pix3[2] <= p2[20]; pix4[2] <=
             p2[21];
250       pix1[3] <= p1[21]; pix2[3] <= p1[22]; pix3[3] <= p2[21]; pix4[3] <=
             p2[22];
251       pix1[4] <= p1[22]; pix2[4] <= p1[23]; pix3[4] <= p2[22]; pix4[4] <=
             p2[23];
252       pix1[5] <= p1[22]; pix2[5] <= p1[23]; pix3[5] <= p2[22]; pix4[5] <=
             p2[23];
253       pix1[6] <= p1[24]; pix2[6] <= p1[25]; pix3[6] <= p2[24]; pix4[6] <=
             p2[25];
254       pix1[7] <= p1[24]; pix2[7] <= p1[25]; pix3[7] <= p2[24]; pix4[7] <=
             p2[25];
255
256       line_count <= line_count + 1;
257
258       // SHIFT IN NEW DATA IN [127:64]
259       reg_shift_r <= 1'b1;
260     end
261
262
263     else begin  // line_count == 4
264
265       pix1[0] <= p1[25]; pix2[0] <= p1[26]; pix3[0] <= p2[25]; pix4[0] <=
             p2[26];
266       pix1[1] <= p1[26]; pix2[1] <= p1[27]; pix3[1] <= p2[26]; pix4[1] <=
             p2[27];
267       pix1[2] <= p1[26]; pix2[2] <= p1[27]; pix3[2] <= p2[26]; pix4[2] <=
             p2[27];
268       pix1[3] <= p1[28]; pix2[3] <= p1[29]; pix3[3] <= p2[28]; pix4[3] <=
```

```
            p2[29];
269       pix1[4] <= p1[28]; pix2[4] <= p1[29]; pix3[4] <= p2[28]; pix4[4] <=
            p2[29];
270       pix1[5] <= p1[29]; pix2[5] <= p1[30]; pix3[5] <= p2[29]; pix4[5] <=
            p2[30];
271       pix1[6] <= p1[30]; pix2[6] <= p1[31]; pix3[6] <= p2[30]; pix4[6] <=
            p2[31];
272       pix1[7] <= p1[30]; pix2[7] <= p1[31]; pix3[7] <= p2[30]; pix4[7] <=
            p2[31];
273
274
275       line_count <= 3'b000;
276
277       // SHIFT IN NEW DATA IN [63:0]
278       reg_shift_r <= 1'b1;
279
280     end
281
282
283     //----------------------------LINE 0-----------------------------------//
284     if (line_number == 3'b000) begin
285       if (line_count == 3'b000) begin
286
287         s1[0] <= a16; s2[0] <= a00; s3[0] <= a00; s4[0] <= a00;
288         s1[1] <= a04; s2[1] <= a12; s3[1] <= a00; s4[1] <= a00;
289         s1[2] <= a08; s2[2] <= a08; s3[2] <= a00; s4[2] <= a00;
290         s1[3] <= a12; s2[3] <= a04; s3[3] <= a00; s4[3] <= a00;
291         s1[4] <= a00; s2[4] <= a16; s3[4] <= a00; s4[4] <= a00;
292         s1[5] <= a16; s2[5] <= a00; s3[5] <= a00; s4[5] <= a00;
293         s1[6] <= a04; s2[6] <= a12; s3[6] <= a00; s4[6] <= a00;
294         s1[7] <= a08; s2[7] <= a08; s3[7] <= a00; s4[7] <= a00;
295
296       end
297
298       else if (line_count == 3'b001) begin
299
300         s1[0] <= a12; s2[0] <= a04; s3[0] <= a00; s4[0] <= a00;
301         s1[1] <= a00; s2[1] <= a16; s3[1] <= a00; s4[1] <= a00;
302         s1[2] <= a16; s2[2] <= a00; s3[2] <= a00; s4[2] <= a00;
303         s1[3] <= a04; s2[3] <= a12; s3[3] <= a00; s4[3] <= a00;
304         s1[4] <= a08; s2[4] <= a08; s3[4] <= a00; s4[4] <= a00;
305         s1[5] <= a12; s2[5] <= a04; s3[5] <= a00; s4[5] <= a00;
306         s1[6] <= a00; s2[6] <= a16; s3[6] <= a00; s4[6] <= a00;
307         s1[7] <= a16; s2[7] <= a00; s3[7] <= a00; s4[7] <= a00;
308
309       end
310
311       else if (line_count == 3'b010) begin
312
313         s1[0] <= a04; s2[0] <= a12; s3[0] <= a00; s4[0] <= a00;
314         s1[1] <= a08; s2[1] <= a08; s3[1] <= a00; s4[1] <= a00;
```

```
315          s1[2] <= a12; s2[2] <= a04; s3[2] <= a00; s4[2] <= a00;
316          s1[3] <= a00; s2[3] <= a16; s3[3] <= a00; s4[3] <= a00;
317          s1[4] <= a16; s2[4] <= a00; s3[4] <= a00; s4[4] <= a00;
318          s1[5] <= a04; s2[5] <= a12; s3[5] <= a00; s4[5] <= a00;
319          s1[6] <= a08; s2[6] <= a08; s3[6] <= a00; s4[6] <= a00;
320          s1[7] <= a12; s2[7] <= a04; s3[7] <= a00; s4[7] <= a00;
321
322        end
323
324        else if (line_count == 3'b011) begin
325
326          s1[0] <= a00; s2[0] <= a16; s3[0] <= a00; s4[0] <= a00;
327          s1[1] <= a16; s2[1] <= a00; s3[1] <= a00; s4[1] <= a00;
328          s1[2] <= a04; s2[2] <= a12; s3[2] <= a00; s4[2] <= a00;
329          s1[3] <= a08; s2[3] <= a08; s3[3] <= a00; s4[3] <= a00;
330          s1[4] <= a12; s2[4] <= a04; s3[4] <= a00; s4[4] <= a00;
331          s1[5] <= a00; s2[5] <= a16; s3[5] <= a00; s4[5] <= a00;
332          s1[6] <= a16; s2[6] <= a00; s3[6] <= a00; s4[6] <= a00;
333          s1[7] <= a04; s2[7] <= a12; s3[7] <= a00; s4[7] <= a00;
334
335        end
336
337        else begin  // line_count == 4
338
339          s1[0] <= a08; s2[0] <= a08; s3[0] <= a00; s4[0] <= a00;
340          s1[1] <= a12; s2[1] <= a04; s3[1] <= a00; s4[1] <= a00;
341          s1[2] <= a00; s2[2] <= a16; s3[2] <= a00; s4[2] <= a00;
342          s1[3] <= a16; s2[3] <= a00; s3[3] <= a00; s4[3] <= a00;
343          s1[4] <= a04; s2[4] <= a12; s3[4] <= a00; s4[4] <= a00;
344          s1[5] <= a08; s2[5] <= a08; s3[5] <= a00; s4[5] <= a00;
345          s1[6] <= a12; s2[6] <= a04; s3[6] <= a00; s4[6] <= a00;
346          s1[7] <= a00; s2[7] <= a16; s3[7] <= a00; s4[7] <= a00;
347
348        end
349      end
350
351      //---------------------------LINE 1-----------------------------------//
352      else if (line_number == 3'b001) begin
353        if (line_count == 3'b000) begin
354
355          s1[0] <= a04; s2[0] <= a00; s3[0] <= a12; s4[0] <= a00;
356          s1[1] <= a01; s2[1] <= a03; s3[1] <= a03; s4[1] <= a09;
357          s1[2] <= a02; s2[2] <= a02; s3[2] <= a06; s4[2] <= a06;
358          s1[3] <= a03; s2[3] <= a01; s3[3] <= a09; s4[3] <= a03;
359          s1[4] <= a00; s2[4] <= a04; s3[4] <= a00; s4[4] <= a12;
360          s1[5] <= a04; s2[5] <= a00; s3[5] <= a12; s4[5] <= a00;
361          s1[6] <= a01; s2[6] <= a03; s3[6] <= a03; s4[6] <= a09;
362          s1[7] <= a02; s2[7] <= a02; s3[7] <= a06; s4[7] <= a06;
363
364        end
365
```

```
366         else if (line_count == 3'b001) begin
367
368           s1[0] <= a03; s2[0] <= a01; s3[0] <= a09; s4[0] <= a03;
369           s1[1] <= a00; s2[1] <= a04; s3[1] <= a00; s4[1] <= a12;
370           s1[2] <= a04; s2[2] <= a00; s3[2] <= a12; s4[2] <= a00;
371           s1[3] <= a01; s2[3] <= a03; s3[3] <= a03; s4[3] <= a09;
372           s1[4] <= a02; s2[4] <= a02; s3[4] <= a06; s4[4] <= a06;
373           s1[5] <= a03; s2[5] <= a01; s3[5] <= a09; s4[5] <= a03;
374           s1[6] <= a00; s2[6] <= a04; s3[6] <= a00; s4[6] <= a12;
375           s1[7] <= a04; s2[7] <= a00; s3[7] <= a12; s4[7] <= a00;
376
377         end
378
379         else if (line_count == 3'b010) begin
380
381           s1[0] <= a01; s2[0] <= a03; s3[0] <= a03; s4[0] <= a09;
382           s1[1] <= a02; s2[1] <= a02; s3[1] <= a06; s4[1] <= a06;
383           s1[2] <= a03; s2[2] <= a01; s3[2] <= a09; s4[2] <= a03;
384           s1[3] <= a00; s2[3] <= a04; s3[3] <= a00; s4[3] <= a12;
385           s1[4] <= a04; s2[4] <= a00; s3[4] <= a12; s4[4] <= a00;
386           s1[5] <= a01; s2[5] <= a03; s3[5] <= a03; s4[5] <= a09;
387           s1[6] <= a02; s2[6] <= a02; s3[6] <= a06; s4[6] <= a06;
388           s1[7] <= a03; s2[7] <= a01; s3[7] <= a09; s4[7] <= a03;
389
390         end
391
392         else if (line_count == 3'b011) begin
393
394           s1[0] <= a00; s2[0] <= a04; s3[0] <= a00; s4[0] <= a12;
395           s1[1] <= a04; s2[1] <= a00; s3[1] <= a12; s4[1] <= a00;
396           s1[2] <= a01; s2[2] <= a03; s3[2] <= a03; s4[2] <= a09;
397           s1[3] <= a02; s2[3] <= a02; s3[3] <= a06; s4[3] <= a06;
398           s1[4] <= a03; s2[4] <= a01; s3[4] <= a09; s4[4] <= a03;
399           s1[5] <= a00; s2[5] <= a04; s3[5] <= a00; s4[5] <= a12;
400           s1[6] <= a04; s2[6] <= a00; s3[6] <= a12; s4[6] <= a00;
401           s1[7] <= a01; s2[7] <= a03; s3[7] <= a03; s4[7] <= a09;
402
403         end
404
405         else begin  // line_count == 4
406
407           s1[0] <= a02; s2[0] <= a02; s3[0] <= a06; s4[0] <= a06;
408           s1[1] <= a03; s2[1] <= a01; s3[1] <= a09; s4[1] <= a03;
409           s1[2] <= a00; s2[2] <= a04; s3[2] <= a00; s4[2] <= a12;
410           s1[3] <= a04; s2[3] <= a00; s3[3] <= a12; s4[3] <= a00;
411           s1[4] <= a01; s2[4] <= a03; s3[4] <= a03; s4[4] <= a09;
412           s1[5] <= a02; s2[5] <= a02; s3[5] <= a06; s4[5] <= a06;
413           s1[6] <= a03; s2[6] <= a01; s3[6] <= a09; s4[6] <= a03;
414           s1[7] <= a00; s2[7] <= a04; s3[7] <= a00; s4[7] <= a12;
415
416         end
```

```
417         end
418
419         //--------------------------LINE 2-----------------------------------//
420         else if (line_number == 3'b010) begin
421           if (line_count == 3'b000) begin
422
423             s1[0] <= a08; s2[0] <= a00; s3[0] <= a08; s4[0] <= a00;
424             s1[1] <= a02; s2[1] <= a06; s3[1] <= a02; s4[1] <= a06;
425             s1[2] <= a04; s2[2] <= a04; s3[2] <= a04; s4[2] <= a04;
426             s1[3] <= a06; s2[3] <= a02; s3[3] <= a06; s4[3] <= a02;
427             s1[4] <= a00; s2[4] <= a08; s3[4] <= a00; s4[4] <= a08;
428             s1[5] <= a08; s2[5] <= a00; s3[5] <= a08; s4[5] <= a00;
429             s1[6] <= a02; s2[6] <= a06; s3[6] <= a02; s4[6] <= a06;
430             s1[7] <= a04; s2[7] <= a04; s3[7] <= a04; s4[7] <= a04;
431
432           end
433
434           else if (line_count == 3'b001) begin
435
436             s1[0] <= a06; s2[0] <= a02; s3[0] <= a06; s4[0] <= a02;
437             s1[1] <= a00; s2[1] <= a08; s3[1] <= a00; s4[1] <= a08;
438             s1[2] <= a08; s2[2] <= a00; s3[2] <= a08; s4[2] <= a00;
439             s1[3] <= a02; s2[3] <= a06; s3[3] <= a02; s4[3] <= a06;
440             s1[4] <= a04; s2[4] <= a04; s3[4] <= a04; s4[4] <= a04;
441             s1[5] <= a06; s2[5] <= a02; s3[5] <= a06; s4[5] <= a02;
442             s1[6] <= a00; s2[6] <= a08; s3[6] <= a00; s4[6] <= a08;
443             s1[7] <= a08; s2[7] <= a00; s3[7] <= a08; s4[7] <= a00;
444
445           end
446
447           else if (line_count == 3'b010) begin
448
449             s1[0] <= a02; s2[0] <= a06; s3[0] <= a02; s4[0] <= a06;
450             s1[1] <= a04; s2[1] <= a04; s3[1] <= a04; s4[1] <= a04;
451             s1[2] <= a06; s2[2] <= a02; s3[2] <= a06; s4[2] <= a02;
452             s1[3] <= a00; s2[3] <= a08; s3[3] <= a00; s4[3] <= a08;
453             s1[4] <= a08; s2[4] <= a00; s3[4] <= a08; s4[4] <= a00;
454             s1[5] <= a02; s2[5] <= a06; s3[5] <= a02; s4[5] <= a06;
455             s1[6] <= a04; s2[6] <= a04; s3[6] <= a04; s4[6] <= a04;
456             s1[7] <= a06; s2[7] <= a02; s3[7] <= a06; s4[7] <= a02;
457
458           end
459
460           else if (line_count == 3'b011) begin
461
462             s1[0] <= a00; s2[0] <= a08; s3[0] <= a00; s4[0] <= a08;
463             s1[1] <= a08; s2[1] <= a00; s3[1] <= a08; s4[1] <= a00;
464             s1[2] <= a02; s2[2] <= a06; s3[2] <= a02; s4[2] <= a06;
465             s1[3] <= a04; s2[3] <= a04; s3[3] <= a04; s4[3] <= a04;
466             s1[4] <= a06; s2[4] <= a02; s3[4] <= a06; s4[4] <= a02;
467             s1[5] <= a00; s2[5] <= a08; s3[5] <= a00; s4[5] <= a08;
```

```
468          s1[6] <= a08; s2[6] <= a00; s3[6] <= a08; s4[6] <= a00;
469          s1[7] <= a02; s2[7] <= a06; s3[7] <= a02; s4[7] <= a06;
470
471        end
472
473        else begin  // line_count == 4
474
475          s1[0] <= a04; s2[0] <= a04; s3[0] <= a04; s4[0] <= a04;
476          s1[1] <= a06; s2[1] <= a02; s3[1] <= a06; s4[1] <= a02;
477          s1[2] <= a00; s2[2] <= a08; s3[2] <= a00; s4[2] <= a08;
478          s1[3] <= a08; s2[3] <= a00; s3[3] <= a08; s4[3] <= a00;
479          s1[4] <= a02; s2[4] <= a06; s3[4] <= a02; s4[4] <= a06;
480          s1[5] <= a04; s2[5] <= a04; s3[5] <= a04; s4[5] <= a04;
481          s1[6] <= a06; s2[6] <= a02; s3[6] <= a06; s4[6] <= a02;
482          s1[7] <= a00; s2[7] <= a08; s3[7] <= a00; s4[7] <= a08;
483
484        end
485      end
486
487      //---------------------------LINE 3-----------------------------------//
488      else if (line_number == 3'b011) begin
489        if (line_count == 3'b000) begin
490
491          s1[0] <= a12; s2[0] <= a00; s3[0] <= a04; s4[0] <= a00;
492          s1[1] <= a03; s2[1] <= a09; s3[1] <= a01; s4[1] <= a03;
493          s1[2] <= a06; s2[2] <= a06; s3[2] <= a02; s4[2] <= a02;
494          s1[3] <= a09; s2[3] <= a03; s3[3] <= a03; s4[3] <= a01;
495          s1[4] <= a00; s2[4] <= a12; s3[4] <= a00; s4[4] <= a04;
496          s1[5] <= a12; s2[5] <= a00; s3[5] <= a04; s4[5] <= a00;
497          s1[6] <= a03; s2[6] <= a09; s3[6] <= a01; s4[6] <= a03;
498          s1[7] <= a06; s2[7] <= a06; s3[7] <= a02; s4[7] <= a02;
499
500        end
501
502        else if (line_count == 3'b001) begin
503
504          s1[0] <= a09; s2[0] <= a03; s3[0] <= a03; s4[0] <= a01;
505          s1[1] <= a00; s2[1] <= a12; s3[1] <= a00; s4[1] <= a04;
506          s1[2] <= a12; s2[2] <= a00; s3[2] <= a04; s4[2] <= a00;
507          s1[3] <= a03; s2[3] <= a09; s3[3] <= a01; s4[3] <= a03;
508          s1[4] <= a06; s2[4] <= a06; s3[4] <= a02; s4[4] <= a02;
509          s1[5] <= a09; s2[5] <= a03; s3[5] <= a03; s4[5] <= a01;
510          s1[6] <= a00; s2[6] <= a12; s3[6] <= a00; s4[6] <= a04;
511          s1[7] <= a12; s2[7] <= a00; s3[7] <= a04; s4[7] <= a00;
512
513        end
514
515        else if (line_count == 3'b010) begin
516
517          s1[0] <= a03; s2[0] <= a09; s3[0] <= a01; s4[0] <= a03;
518          s1[1] <= a06; s2[1] <= a06; s3[1] <= a02; s4[1] <= a02;
```

```
519           s1[2] <= a09; s2[2] <= a03; s3[2] <= a03; s4[2] <= a01;
520           s1[3] <= a00; s2[3] <= a12; s3[3] <= a00; s4[3] <= a04;
521           s1[4] <= a12; s2[4] <= a00; s3[4] <= a04; s4[4] <= a00;
522           s1[5] <= a03; s2[5] <= a09; s3[5] <= a01; s4[5] <= a03;
523           s1[6] <= a06; s2[6] <= a06; s3[6] <= a02; s4[6] <= a02;
524           s1[7] <= a09; s2[7] <= a03; s3[7] <= a03; s4[7] <= a01;
525
526         end
527
528         else if (line_count == 3'b011) begin
529
530           s1[0] <= a00; s2[0] <= a12; s3[0] <= a00; s4[0] <= a04;
531           s1[1] <= a12; s2[1] <= a00; s3[1] <= a04; s4[1] <= a00;
532           s1[2] <= a03; s2[2] <= a09; s3[2] <= a01; s4[2] <= a03;
533           s1[3] <= a06; s2[3] <= a06; s3[3] <= a02; s4[3] <= a02;
534           s1[4] <= a09; s2[4] <= a03; s3[4] <= a03; s4[4] <= a01;
535           s1[5] <= a00; s2[5] <= a12; s3[5] <= a00; s4[5] <= a04;
536           s1[6] <= a12; s2[6] <= a00; s3[6] <= a04; s4[6] <= a00;
537           s1[7] <= a03; s2[7] <= a09; s3[7] <= a01; s4[7] <= a03;
538
539         end
540
541         else begin  // line_count == 4
542
543           s1[0] <= a06; s2[0] <= a06; s3[0] <= a02; s4[0] <= a02;
544           s1[1] <= a09; s2[1] <= a03; s3[1] <= a03; s4[1] <= a01;
545           s1[2] <= a00; s2[2] <= a12; s3[2] <= a00; s4[2] <= a04;
546           s1[3] <= a12; s2[3] <= a00; s3[3] <= a04; s4[3] <= a00;
547           s1[4] <= a03; s2[4] <= a09; s3[4] <= a01; s4[4] <= a03;
548           s1[5] <= a06; s2[5] <= a06; s3[5] <= a02; s4[5] <= a02;
549           s1[6] <= a09; s2[6] <= a03; s3[6] <= a03; s4[6] <= a01;
550           s1[7] <= a00; s2[7] <= a12; s3[7] <= a00; s4[7] <= a04;
551
552         end
553       end
554
555       //---------------------------LINE 4-----------------------------------//
556       else begin // line_number == 4
557         if (line_count == 3'b000) begin
558
559           s1[0] <= a00; s2[0] <= a00; s3[0] <= a16; s4[0] <= a00;
560           s1[1] <= a00; s2[1] <= a00; s3[1] <= a04; s4[1] <= a12;
561           s1[2] <= a00; s2[2] <= a00; s3[2] <= a08; s4[2] <= a08;
562           s1[3] <= a00; s2[3] <= a00; s3[3] <= a12; s4[3] <= a04;
563           s1[4] <= a00; s2[4] <= a00; s3[4] <= a00; s4[4] <= a16;
564           s1[5] <= a00; s2[5] <= a00; s3[5] <= a16; s4[5] <= a00;
565           s1[6] <= a00; s2[6] <= a00; s3[6] <= a04; s4[6] <= a12;
566           s1[7] <= a00; s2[7] <= a00; s3[7] <= a08; s4[7] <= a08;
567
568         end
569
```

```
570         else if (line_count == 3'b001) begin
571
572           s1[0] <= a00; s2[0] <= a00; s3[0] <= a12; s4[0] <= a04;
573           s1[1] <= a00; s2[1] <= a00; s3[1] <= a00; s4[1] <= a16;
574           s1[2] <= a00; s2[2] <= a00; s3[2] <= a16; s4[2] <= a00;
575           s1[3] <= a00; s2[3] <= a00; s3[3] <= a04; s4[3] <= a12;
576           s1[4] <= a00; s2[4] <= a00; s3[4] <= a08; s4[4] <= a08;
577           s1[5] <= a00; s2[5] <= a00; s3[5] <= a12; s4[5] <= a04;
578           s1[6] <= a00; s2[6] <= a00; s3[6] <= a00; s4[6] <= a16;
579           s1[7] <= a00; s2[7] <= a00; s3[7] <= a16; s4[7] <= a00;
580
581         end
582
583         else if (line_count == 3'b010) begin
584
585           s1[0] <= a00; s2[0] <= a00; s3[0] <= a04; s4[0] <= a12;
586           s1[1] <= a00; s2[1] <= a00; s3[1] <= a08; s4[1] <= a08;
587           s1[2] <= a00; s2[2] <= a00; s3[2] <= a12; s4[2] <= a04;
588           s1[3] <= a00; s2[3] <= a00; s3[3] <= a00; s4[3] <= a16;
589           s1[4] <= a00; s2[4] <= a00; s3[4] <= a16; s4[4] <= a00;
590           s1[5] <= a00; s2[5] <= a00; s3[5] <= a04; s4[5] <= a12;
591           s1[6] <= a00; s2[6] <= a00; s3[6] <= a08; s4[6] <= a08;
592           s1[7] <= a00; s2[7] <= a00; s3[7] <= a12; s4[7] <= a04;
593
594         end
595
596         else if (line_count == 3'b011) begin
597
598           s1[0] <= a00; s2[0] <= a00; s3[0] <= a00; s4[0] <= a16;
599           s1[1] <= a00; s2[1] <= a00; s3[1] <= a16; s4[1] <= a00;
600           s1[2] <= a00; s2[2] <= a00; s3[2] <= a04; s4[2] <= a12;
601           s1[3] <= a00; s2[3] <= a00; s3[3] <= a08; s4[3] <= a08;
602           s1[4] <= a00; s2[4] <= a00; s3[4] <= a12; s4[4] <= a04;
603           s1[5] <= a00; s2[5] <= a00; s3[5] <= a00; s4[5] <= a16;
604           s1[6] <= a00; s2[6] <= a00; s3[6] <= a16; s4[6] <= a00;
605           s1[7] <= a00; s2[7] <= a00; s3[7] <= a04; s4[7] <= a12;
606
607         end
608
609         else begin  // line_count == 4
610
611           s1[0] <= a00; s2[0] <= a00; s3[0] <= a08; s4[0] <= a08;
612           s1[1] <= a00; s2[1] <= a00; s3[1] <= a12; s4[1] <= a04;
613           s1[2] <= a00; s2[2] <= a00; s3[2] <= a00; s4[2] <= a16;
614           s1[3] <= a00; s2[3] <= a00; s3[3] <= a16; s4[3] <= a00;
615           s1[4] <= a00; s2[4] <= a00; s3[4] <= a04; s4[4] <= a12;
616           s1[5] <= a00; s2[5] <= a00; s3[5] <= a08; s4[5] <= a08;
617           s1[6] <= a00; s2[6] <= a00; s3[6] <= a12; s4[6] <= a04;
618           s1[7] <= a00; s2[7] <= a00; s3[7] <= a00; s4[7] <= a16;
619
620         end
```

```
621        end
622
623        //------------------------ Check horizontal -------------------------//
624        if (horizontal_count == FRAME_WIDTH_OUT-1) begin // A line is finished.
625          horizontal_count <= 0;
626          if (line_number == 4) begin
627            line_number <= 0;
628          end
629          else
630            line_number <= line_number + 1;
631
632          //------------------------ Check Vertical -------------------------//
633          if (vertical_count == FRAME_HEIGHT_OUT-1) begin
634            vertical_count <= 0;
635            frame_finished_r <= 1'b1;
636            if (yuv_count == 2)
637              yuv_count <= 0;
638            else
639              yuv_count <= yuv_count + 1;
640          end
641          else
642            vertical_count <= vertical_count + 1;
643
644        end
645        else
646          horizontal_count <= horizontal_count + 1;
647
648      end // if (FIFO is not full AND registers are ready)
649
650      if (wr_en_r)
651        dummy_count <= dummy_count + 1; // used for simulation
652
653      if ((horizontal_count == FRAME_WIDTH_OUT-1) &&
654          (vertical_count == FRAME_HEIGHT_OUT-1)) begin
655        if (yuv_count == 2)
656          last_data_packet <= 1;
657      end
658
659    end // elsif(posedge ACLK)
660  end // always
661
662
663  always @(posedge FIFO_full) begin
664  dummy_FIFO_full_count <= dummy_FIFO_full_count + 1; // used for simulation
665  end
666  /***************************************************************************
667  * Drive outputs
668  ***************************************************************************/
669  assign din            = fifo_data;
670  assign wr_en          = wr_en_r;
671  assign frame_finished = last_data_packet; //frame_finished_r
```

```
672 assign reg_shift        = reg_shift_r;
673
674 endmodule
```

B.6 Scale 1.875 module

```
1  /*****************************************************************************/
2  /*                                                                           */
3  /* Author: Roger Skarboe                                                     */
4  /*                                                                           */
5  /* Version: v 1.06, 01.06.2010                                               */
6  /*                                                                           */
7  /* Description: Module to scale from SD to FullHD (scale factor: 1.875).     */
8  /*              Module gets input from AXI Read module.                      */
9  /*              Module gets input from cooperation register module.          */
10 /*                                                                           */
11 /*****************************************************************************/
12
13 module scale1875(ACLK, ARESETn, din, FIFO_full, wr_en, frame_finished,
14   reg_ready, reg_shift, line1_r, line2_r, frame_width_o, frame_height_o);
15
16 /* Global input signals */
17 input          ACLK;
18 input          ARESETn;
19
20 /* FIFO output signals */
21 output [63:0]  din; // Assign to data in channel on FIFO
22 output         wr_en;
23 input          FIFO_full; // Assign FIFOs almost full
24
25
26 /* AXI Write signals */
27 output         frame_finished; // Used for IRQ on AXI Write
28
29 /* Register signals */
30 input          reg_ready; // Tell if registers har data ready
31 output         reg_shift; // Oppdate registers
32
33 /* APB signals */
34 input  [10:0]  frame_width_o;
35 input  [10:0]  frame_height_o;
36
37 /******************************************************************************
38 * Register space
39 ******************************************************************************/
40 input [255:0]   line1_r, line2_r;
41 reg             wr_en_r;
42 reg             frame_finished_r;
43 reg             reg_shift_r;
44 reg   [63:0]    fifo_data;
45 wire  [7:0]     FRAME_WIDTH_OUT;
46 wire  [10:0]    FRAME_HEIGHT_OUT;
47
48
49 // Counters:
```

```
50 reg   [3:0]      line_number; // 5 line number used. Toggle between 5 lines.
51 reg   [3:0]      line_count;  // 5 different ways to calculate output per line.
52 reg   [8:0]      horizontal_count; // +1 for every 5 output pixels. Max: 256
53 reg   [10:0]     vertical_count; // +1 for every 5 output lines in frame. (144)
54 reg   [1:0]      yuv_count;
55
56 reg   [3:0]      x_count;
57 reg   [3:0]      y_count;
58 reg   [3:0]      y_reg;
59
60 reg   [15:0]     dummy_count;
61 reg              last_data_packet; // Ensure that last data is read.
62
63 // Input register for task (odd_func):
64 reg [3:0] x_r [0:7]; // changed from 4:0...
65
66 reg [7:0] pix1 [0:7];
67 reg [7:0] pix2 [0:7];
68 reg [7:0] pix3 [0:7];
69 reg [7:0] pix4 [0:7];
70
71 // Wire p1 and p2 to line1 and line2 register:
72 wire [7:0] p1 [0:31];
73 wire [7:0] p2 [0:31];
74
75 assign {p1[7], p1[6], p1[5], p1[4], p1[3], p1[2], p1[1], p1[0], p1[15], p1[14],
76 p1[13], p1[12], p1[11], p1[10], p1[9], p1[8], p1[23], p1[22], p1[21], p1[20],
77 p1[19], p1[18], p1[17], p1[16], p1[31], p1[30], p1[29], p1[28], p1[27], p1[26],
78 p1[25], p1[24]} = line1_r;
79
80 assign {p2[7], p2[6], p2[5], p2[4], p2[3], p2[2], p2[1], p2[0], p2[15], p2[14],
81 p2[13], p2[12], p2[11], p2[10], p2[9], p2[8], p2[23], p2[22], p2[21], p2[20],
82 p2[19], p2[18], p2[17], p2[16], p2[31], p2[30], p2[29], p2[28], p2[27], p2[26],
83 p2[25], p2[24]} = line2_r;
84
85
86 /* Assign constants */
87 assign FRAME_WIDTH_OUT  = frame_width_o [10:3];
88 assign FRAME_HEIGHT_OUT = frame_height_o;
89
90
91 /*******************************************************************************
92 * Function / Task declaration
93 *******************************************************************************/
94
95 task odd_func;
96   input [3:0] x_r,y_r; // max 4'b1110 in
97   input [7:0] pix1_r,pix2_r,pix3_r,pix4_r; // pixel value
98   output [7:0] po;
99
100  reg [3:0] x_plus1;
```

```
101  reg [3:0] y_plus1;
102  reg [2:0] x_plus1_div2;
103  reg [2:0] y_plus1_div2;
104  reg [5:0] x_plus1_mult4;
105  reg [5:0] y_plus1_mult4;
106  reg [6:0] x_mult_y_div2;
107  reg [3:0] eight_minus_x_plus1_div2;
108  reg [3:0] eight_minus_y_plus1_div2;
109
110  reg [7:0] po_when_both_modulo;
111
112  reg [10:0] modulo_arg1;
113  reg [7:0] modulo_arg_pix1;
114  reg [7:0] modulo_arg_pix2;
115  reg [3:0] modulo_arg_8minus;
116  reg [2:0] modulo_arg_x_or_y;
117
118  reg [13:0] temp1, temp2, temp3, temp4;
119  reg [13:0] temp_res;
120
121  begin
122
123    // Temporary registers:
124    x_plus1 = x_r + 1;                              // x+1
125    y_plus1 = y_r + 1;                              // y+1
126
127    x_plus1_div2 = x_plus1 [3:1];                   // (x+1)/2
128    y_plus1_div2 = y_plus1 [3:1];                   // (y+1)/2
129
130    x_plus1_mult4 = {x_plus1, 2'b0};                // (x+1)*4
131    y_plus1_mult4 = {y_plus1, 2'b0};                // (y+1)*4
132
133    eight_minus_x_plus1_div2 = 8 - x_plus1_div2; // 8-(x+1)/2
134    eight_minus_y_plus1_div2 = 8 - y_plus1_div2; // 8-(y+1)/2
135
136    x_mult_y_div2 = x_plus1_div2 * y_plus1_div2; // ((x+1)/2)*((y+1)/2)
137
138    // Set the right registers when x and y is even:
139    if (y_r == 14)
140      po_when_both_modulo = pix3_r;
141    else
142      po_when_both_modulo = pix1_r;
143
144
145    // Set registers when x or y is even:
146    if (y_r[0] == 0) begin
147      if (y_r == 14) begin
148        modulo_arg_pix1 = pix3_r;
149        modulo_arg_pix2 = pix4_r;
150      end
151      else begin
```

```
152         modulo_arg_pix1 = pix1_r;
153         modulo_arg_pix2 = pix2_r;
154       end
155       modulo_arg_8minus = eight_minus_x_plus1_div2;
156       modulo_arg_x_or_y = x_plus1_div2;
157     end
158
159     else begin
160       modulo_arg_pix1 = pix1_r;
161       modulo_arg_pix2 = pix3_r;
162       modulo_arg_8minus = eight_minus_y_plus1_div2;
163       modulo_arg_x_or_y = y_plus1_div2;
164     end
165
166     // This is use for both averaging with 2 and 4 pixels:
167     // (x or y is even) or (x and y is odd)
168     modulo_arg1 = modulo_arg_8minus * modulo_arg_pix1;
169
170     // Temp registers for when x and y is odd:
171     temp4 = x_mult_y_div2 * pix4_r;
172     temp3 = (y_plus1_mult4 - x_mult_y_div2) * pix3_r;
173     temp2 = (x_plus1_mult4 - x_mult_y_div2) * pix2_r;
174     temp1 = eight_minus_x_plus1_div2 * modulo_arg1;
175
176
177     // Determin output value:
178     if (x_r[0] == 0 && y_r[0] == 0) // (x_r and y_r are even)
179       po = po_when_both_modulo;
180
181     else if (y_r[0] == 0 || x_r[0] == 0) begin // (x_r or y_r are even)
182       temp_res = modulo_arg1 + (modulo_arg_x_or_y * modulo_arg_pix2);
183       po = temp_res [10:3];
184     end
185
186     else begin // (x_r and y_r are odd)
187       temp_res = temp1 + temp2 + temp3 + temp4;
188       po = temp_res[13:6];
189     end
190   end
191 endtask
192
193
194
195 /* Process */
196 always @(pix1[0] or pix1[1] or pix1[2] or pix1[3] or pix1[4] or pix1[5] or
197   pix1[6] or pix1[7] or pix2[0] or pix2[1] or pix2[2] or pix2[3] or pix2[4] or
198   pix2[5] or pix2[6] or pix2[7] or pix3[0] or pix3[1] or pix3[2] or pix3[3] or
199   pix3[4] or pix3[5] or pix3[6] or pix3[7] or pix4[0] or pix4[1] or pix4[2] or
200   pix4[3] or pix4[4] or pix4[5] or pix4[6] or pix4[7] or x_r[0] or x_r[1] or
201   x_r[2] or x_r[3] or x_r[4] or x_r[5] or x_r[6] or x_r[7] or y_reg)
202 begin
```

```
203
204   odd_func(x_r[0], y_reg, pix1[0],pix2[0],pix3[0],pix4[0], fifo_data [7:0]);
205   odd_func(x_r[1], y_reg, pix1[1],pix2[1],pix3[1],pix4[1], fifo_data [15:8]);
206   odd_func(x_r[2], y_reg, pix1[2],pix2[2],pix3[2],pix4[2], fifo_data [23:16]);
207   odd_func(x_r[3], y_reg, pix1[3],pix2[3],pix3[3],pix4[3], fifo_data [31:24]);
208   odd_func(x_r[4], y_reg, pix1[4],pix2[4],pix3[4],pix4[4], fifo_data [39:32]);
209   odd_func(x_r[5], y_reg, pix1[5],pix2[5],pix3[5],pix4[5], fifo_data [47:40]);
210   odd_func(x_r[6], y_reg, pix1[6],pix2[6],pix3[6],pix4[6], fifo_data [55:48]);
211   odd_func(x_r[7], y_reg, pix1[7],pix2[7],pix3[7],pix4[7], fifo_data [63:56]);
212
213 end
214
215
216
217 /* Process */
218 always @(negedge ARESETn or posedge ACLK)
219 begin
220   if (!ARESETn)
221   begin
222     // Output registers:
223     reg_shift_r <= 1'b0;
224     frame_finished_r <= 1'b0;
225     wr_en_r <= 1'b0;
226     // Counters:
227     horizontal_count <= 0;
228     vertical_count <= 0;
229     line_number <= 3'b000;
230     line_count <= 3'b000;
231
232     last_data_packet <= 0;
233     dummy_count <= 0;
234     yuv_count <= 0;
235     y_count <= 0;
236     y_reg <= 0;
237     // Reset intern registers? (+ 123 LUTS (20 % more), + 14MHz (4,3%))
238     //s1 <= 0; s2 <= 0; s3 <= 0; s4 <= 0;
239     //pix1 <= 0; pix2 <= 0; pix3 <= 0; pix4 <= 0;
240   end
241
242   // elsif(posedge ACLK)
243   else
244   begin
245
246     frame_finished_r <= 1'b0; // Default value
247     if (FIFO_full)
248       reg_shift_r <= reg_shift_r; // Hold value when FIFO is full
249     else
250       reg_shift_r <= 1'b1; // Default value
251
252     wr_en_r <= 1'b0; // Defaukt value
253
```

```
254     // Start
255     if (reg_ready || last_data_packet)
256     begin
257
258       wr_en_r <= 1'b1;
259       last_data_packet <= 0;
260
261       line_count <= line_count + 1;
262       y_reg <= y_count; // hold previous value of y_count in y_reg
263
264       case(line_count)
265         4'b0000: begin
266         pix1[0] <= p1[0]; pix2[0] <= p1[1];
267         pix1[1] <= p1[0]; pix2[1] <= p1[1];
268         pix1[2] <= p1[1]; pix2[2] <= p1[2];
269         pix1[3] <= p1[1]; pix2[3] <= p1[2];
270         pix1[4] <= p1[2]; pix2[4] <= p1[3];
271         pix1[5] <= p1[2]; pix2[5] <= p1[3];
272         pix1[6] <= p1[3]; pix2[6] <= p1[4];
273         pix1[7] <= p1[3]; pix2[7] <= p1[4];
274
275         pix3[0] <= p2[0]; pix4[0] <= p2[1];
276         pix3[1] <= p2[0]; pix4[1] <= p2[1];
277         pix3[2] <= p2[1]; pix4[2] <= p2[2];
278         pix3[3] <= p2[1]; pix4[3] <= p2[2];
279         pix3[4] <= p2[2]; pix4[4] <= p2[3];
280         pix3[5] <= p2[2]; pix4[5] <= p2[3];
281         pix3[6] <= p2[3]; pix4[6] <= p2[4];
282         pix3[7] <= p2[3]; pix4[7] <= p2[4]; // 8
283
284         x_r[0] = 0; x_r[1] = 1; x_r[2] = 2; x_r[3] = 3;
285         x_r[4] = 4; x_r[5] = 5; x_r[6] = 6; x_r[7] = 7;
286
287         reg_shift_r <= 1'b0;
288         end
289
290         4'b0001: begin
291         pix1[0] <= p1[4]; pix2[0] <= p1[5];
292         pix1[1] <= p1[4]; pix2[1] <= p1[5];
293         pix1[2] <= p1[5]; pix2[2] <= p1[6];
294         pix1[3] <= p1[5]; pix2[3] <= p1[6];
295         pix1[4] <= p1[6]; pix2[4] <= p1[7];
296         pix1[5] <= p1[6]; pix2[5] <= p1[7];
297         pix1[6] <= p1[7]; pix2[6] <= p1[8];
298         pix1[7] <= p1[8]; pix2[7] <= p1[9];
299
300         pix3[0] <= p2[4]; pix4[0] <= p2[5];
301         pix3[1] <= p2[4]; pix4[1] <= p2[5];
302         pix3[2] <= p2[5]; pix4[2] <= p2[6];
303         pix3[3] <= p2[5]; pix4[3] <= p2[6];
304         pix3[4] <= p2[6]; pix4[4] <= p2[7];
```

```
305          pix3[5] <= p2[6]; pix4[5] <= p2[7];
306          pix3[6] <= p2[7]; pix4[6] <= p2[8]; // 15
307          pix3[7] <= p2[8]; pix4[7] <= p2[9]; // 16
308
309          x_r[0] = 8; x_r[1] = 9; x_r[2] = 10; x_r[3] = 11;
310          x_r[4] = 12; x_r[5] = 13; x_r[6] = 14; x_r[7] = 0;
311
312          // SHIFT IN NEW DATA IN [255:192]
313          reg_shift_r <= 1'b1;
314          end
315
316          4'b0010: begin
317          pix1[0] <= p1[8]; pix2[0] <= p1[9];
318          pix1[1] <= p1[9]; pix2[1] <= p1[10];
319          pix1[2] <= p1[9]; pix2[2] <= p1[10];
320          pix1[3] <= p1[10]; pix2[3] <= p1[11];
321          pix1[4] <= p1[10]; pix2[4] <= p1[11];
322          pix1[5] <= p1[11]; pix2[5] <= p1[12];
323          pix1[6] <= p1[11]; pix2[6] <= p1[12];
324          pix1[7] <= p1[12]; pix2[7] <= p1[13];
325
326          pix3[0] <= p2[8]; pix4[0] <= p2[9];
327          pix3[1] <= p2[9]; pix4[1] <= p2[10];
328          pix3[2] <= p2[9]; pix4[2] <= p2[10];
329          pix3[3] <= p2[10]; pix4[3] <= p2[11];
330          pix3[4] <= p2[10]; pix4[4] <= p2[11];
331          pix3[5] <= p2[11]; pix4[5] <= p2[12];
332          pix3[6] <= p2[11]; pix4[6] <= p2[12];
333          pix3[7] <= p2[12]; pix4[7] <= p2[13]; // 24
334
335          x_r[0] = 1; x_r[1] = 2; x_r[2] = 3; x_r[3] = 4;
336          x_r[4] = 5; x_r[5] = 6; x_r[6] = 7; x_r[7] = 8;
337
338          reg_shift_r <= 1'b0;
339          end
340
341
342          4'b0011: begin
343          pix1[0] <= p1[12]; pix2[0] <= p1[13];
344          pix1[1] <= p1[13]; pix2[1] <= p1[14];
345          pix1[2] <= p1[13]; pix2[2] <= p1[14];
346          pix1[3] <= p1[14]; pix2[3] <= p1[15];
347          pix1[4] <= p1[14]; pix2[4] <= p1[15];
348          pix1[5] <= p1[15]; pix2[5] <= p1[16];
349          pix1[6] <= p1[16]; pix2[6] <= p1[17];
350          pix1[7] <= p1[16]; pix2[7] <= p1[17];
351
352          pix3[0] <= p2[12]; pix4[0] <= p2[13];
353          pix3[1] <= p2[13]; pix4[1] <= p2[14];
354          pix3[2] <= p2[13]; pix4[2] <= p2[14];
355          pix3[3] <= p2[14]; pix4[3] <= p2[15];
```

```
356         pix3[4] <= p2[14]; pix4[4] <= p2[15];
357         pix3[5] <= p2[15]; pix4[5] <= p2[16]; // 30
358         pix3[6] <= p2[16]; pix4[6] <= p2[17];
359         pix3[7] <= p2[16]; pix4[7] <= p2[17]; // 32
360 
361         x_r[0] = 9; x_r[1] = 10; x_r[2] = 11; x_r[3] = 12;
362         x_r[4] = 13; x_r[5] = 14; x_r[6] = 0; x_r[7] = 1;
363 
364         // SHIFT IN NEW DATA IN [191:128]
365         reg_shift_r <= 1'b1;
366         end
367 
368         4'b0100: begin
369         pix1[0] <= p1[17]; pix2[0] <= p1[18];
370         pix1[1] <= p1[17]; pix2[1] <= p1[18];
371         pix1[2] <= p1[18]; pix2[2] <= p1[19];
372         pix1[3] <= p1[18]; pix2[3] <= p1[19];
373         pix1[4] <= p1[19]; pix2[4] <= p1[20];
374         pix1[5] <= p1[19]; pix2[5] <= p1[20];
375         pix1[6] <= p1[20]; pix2[6] <= p1[21];
376         pix1[7] <= p1[20]; pix2[7] <= p1[21];
377 
378         pix3[0] <= p2[17]; pix4[0] <= p2[18];
379         pix3[1] <= p2[17]; pix4[1] <= p2[18];
380         pix3[2] <= p2[18]; pix4[2] <= p2[19];
381         pix3[3] <= p2[18]; pix4[3] <= p2[19];
382         pix3[4] <= p2[19]; pix4[4] <= p2[20];
383         pix3[5] <= p2[19]; pix4[5] <= p2[20];
384         pix3[6] <= p2[20]; pix4[6] <= p2[21];
385         pix3[7] <= p2[20]; pix4[7] <= p2[21]; // 40
386 
387         x_r[0] = 2; x_r[1] = 3; x_r[2] = 4; x_r[3] = 5;
388         x_r[4] = 6; x_r[5] = 7; x_r[6] = 8; x_r[7] = 9;
389 
390         reg_shift_r <= 1'b0;
391         end
392 
393         4'b0101: begin
394         pix1[0] <= p1[21]; pix2[0] <= p1[22];
395         pix1[1] <= p1[21]; pix2[1] <= p1[22];
396         pix1[2] <= p1[22]; pix2[2] <= p1[23];
397         pix1[3] <= p1[22]; pix2[3] <= p1[23];
398         pix1[4] <= p1[23]; pix2[4] <= p1[24];
399         pix1[5] <= p1[24]; pix2[5] <= p1[25];
400         pix1[6] <= p1[24]; pix2[6] <= p1[25];
401         pix1[7] <= p1[25]; pix2[7] <= p1[26];
402 
403         pix3[0] <= p2[21]; pix4[0] <= p2[22];
404         pix3[1] <= p2[21]; pix4[1] <= p2[22];
405         pix3[2] <= p2[22]; pix4[2] <= p2[23];
406         pix3[3] <= p2[22]; pix4[3] <= p2[23];
```

```
407         pix3[4] <= p2[23]; pix4[4] <= p2[24]; // 45
408         pix3[5] <= p2[24]; pix4[5] <= p2[25];
409         pix3[6] <= p2[24]; pix4[6] <= p2[25];
410         pix3[7] <= p2[25]; pix4[7] <= p2[26]; // 48
411
412         x_r[0] = 10; x_r[1] = 11; x_r[2] = 12; x_r[3] = 13;
413         x_r[4] = 14; x_r[5] = 0; x_r[6] = 1; x_r[7] = 2;
414
415         // SHIFT IN NEW DATA IN [127:64]
416         reg_shift_r <= 1'b1;
417         end
418
419         4'b0110: begin
420         pix1[0] <= p1[25]; pix2[0] <= p1[26];
421         pix1[1] <= p1[26]; pix2[1] <= p1[27];
422         pix1[2] <= p1[26]; pix2[2] <= p1[27];
423         pix1[3] <= p1[27]; pix2[3] <= p1[28];
424         pix1[4] <= p1[27]; pix2[4] <= p1[28];
425         pix1[5] <= p1[28]; pix2[5] <= p1[29];
426         pix1[6] <= p1[28]; pix2[6] <= p1[29];
427         pix1[7] <= p1[29]; pix2[7] <= p1[30];
428
429         pix3[0] <= p2[25]; pix4[0] <= p2[26];
430         pix3[1] <= p2[26]; pix4[1] <= p2[27];
431         pix3[2] <= p2[26]; pix4[2] <= p2[27];
432         pix3[3] <= p2[27]; pix4[3] <= p2[28];
433         pix3[4] <= p2[27]; pix4[4] <= p2[28];
434         pix3[5] <= p2[28]; pix4[5] <= p2[29];
435         pix3[6] <= p2[28]; pix4[6] <= p2[29];
436         pix3[7] <= p2[29]; pix4[7] <= p2[30]; // 56
437
438         x_r[0] = 3; x_r[1] = 4; x_r[2] = 5; x_r[3] = 6;
439         x_r[4] = 7; x_r[5] = 8; x_r[6] = 9; x_r[7] = 10;
440
441         reg_shift_r <= 1'b0;
442         end
443
444         4'b0111: begin // 7
445         pix1[0] <= p1[29]; pix2[0] <= p1[30];
446         pix1[1] <= p1[30]; pix2[1] <= p1[31];
447         pix1[2] <= p1[30]; pix2[2] <= p1[31];
448         pix1[3] <= p1[31]; pix2[3] <= p1[0];
449         pix1[4] <= p1[0]; pix2[4] <= p1[1];
450         pix1[5] <= p1[0]; pix2[5] <= p1[1];
451         pix1[6] <= p1[1]; pix2[6] <= p1[2];
452         pix1[7] <= p1[1]; pix2[7] <= p1[2];
453
454         pix3[0] <= p2[29]; pix4[0] <= p2[30];
455         pix3[1] <= p2[30]; pix4[1] <= p2[31];
456         pix3[2] <= p2[30]; pix4[2] <= p2[31];
457         pix3[3] <= p2[31]; pix4[3] <= p2[0]; // 60
```

```
458         pix3[4] <= p2[0]; pix4[4] <= p2[1];
459         pix3[5] <= p2[0]; pix4[5] <= p2[1];
460         pix3[6] <= p2[1]; pix4[6] <= p2[2];
461         pix3[7] <= p2[1]; pix4[7] <= p2[2]; // 64
462
463         x_r[0] = 11; x_r[1] = 12; x_r[2] = 13; x_r[3] = 14;
464         x_r[4] = 0; x_r[5] = 1; x_r[6] = 2; x_r[7] = 3;
465
466         // SHIFT IN NEW DATA IN [63:0]
467         reg_shift_r <= 1'b1;
468         end
469
470
471         4'b1000: begin
472         pix1[0] <= p1[2]; pix2[0] <= p1[3];
473         pix1[1] <= p1[2]; pix2[1] <= p1[3];
474         pix1[2] <= p1[3]; pix2[2] <= p1[4];
475         pix1[3] <= p1[3]; pix2[3] <= p1[4];
476         pix1[4] <= p1[4]; pix2[4] <= p1[5];
477         pix1[5] <= p1[4]; pix2[5] <= p1[5];
478         pix1[6] <= p1[5]; pix2[6] <= p1[6];
479         pix1[7] <= p1[5]; pix2[7] <= p1[6];
480
481         pix3[0] <= p2[2]; pix4[0] <= p2[3];
482         pix3[1] <= p2[2]; pix4[1] <= p2[3];
483         pix3[2] <= p2[3]; pix4[2] <= p2[4];
484         pix3[3] <= p2[3]; pix4[3] <= p2[4];
485         pix3[4] <= p2[4]; pix4[4] <= p2[5];
486         pix3[5] <= p2[4]; pix4[5] <= p2[5];
487         pix3[6] <= p2[5]; pix4[6] <= p2[6];
488         pix3[7] <= p2[5]; pix4[7] <= p2[6]; // 72
489
490         x_r[0] = 4; x_r[1] = 5; x_r[2] = 6; x_r[3] = 7;
491         x_r[4] = 8; x_r[5] = 9; x_r[6] = 10; x_r[7] = 11;
492
493         reg_shift_r <= 1'b0;
494         end
495
496         4'b1001: begin
497         pix1[0] <= p1[6]; pix2[0] <= p1[7];
498         pix1[1] <= p1[6]; pix2[1] <= p1[7];
499         pix1[2] <= p1[7]; pix2[2] <= p1[8];
500         pix1[3] <= p1[8]; pix2[3] <= p1[9];
501         pix1[4] <= p1[8]; pix2[4] <= p1[9];
502         pix1[5] <= p1[9]; pix2[5] <= p1[10];
503         pix1[6] <= p1[9]; pix2[6] <= p1[10];
504         pix1[7] <= p1[10]; pix2[7] <= p1[11];
505
506         pix3[0] <= p2[6]; pix4[0] <= p2[7];
507         pix3[1] <= p2[6]; pix4[1] <= p2[7];
508         pix3[2] <= p2[7]; pix4[2] <= p2[8]; // 75
```

```
509        pix3[3] <= p2[8]; pix4[3] <= p2[9];
510        pix3[4] <= p2[8]; pix4[4] <= p2[9];
511        pix3[5] <= p2[9]; pix4[5] <= p2[10];
512        pix3[6] <= p2[9]; pix4[6] <= p2[10];
513        pix3[7] <= p2[10]; pix4[7] <= p2[11]; // 80
514
515        x_r[0] = 12; x_r[1] = 13; x_r[2] = 14; x_r[3] = 0;
516        x_r[4] = 1; x_r[5] = 2; x_r[6] = 3; x_r[7] = 4;
517
518        // SHIFT IN NEW DATA IN [255:192]
519        reg_shift_r <= 1'b1;
520        end
521
522        4'b1010: begin
523        pix1[0] <= p1[10]; pix2[0] <= p1[11];
524        pix1[1] <= p1[11]; pix2[1] <= p1[12];
525        pix1[2] <= p1[11]; pix2[2] <= p1[12];
526        pix1[3] <= p1[12]; pix2[3] <= p1[13];
527        pix1[4] <= p1[12]; pix2[4] <= p1[13];
528        pix1[5] <= p1[13]; pix2[5] <= p1[14];
529        pix1[6] <= p1[13]; pix2[6] <= p1[14];
530        pix1[7] <= p1[14]; pix2[7] <= p1[15];
531
532        pix3[0] <= p2[10]; pix4[0] <= p2[11];
533        pix3[1] <= p2[11]; pix4[1] <= p2[12];
534        pix3[2] <= p2[11]; pix4[2] <= p2[12];
535        pix3[3] <= p2[12]; pix4[3] <= p2[13];
536        pix3[4] <= p2[12]; pix4[4] <= p2[13];
537        pix3[5] <= p2[13]; pix4[5] <= p2[14];
538        pix3[6] <= p2[13]; pix4[6] <= p2[14];
539        pix3[7] <= p2[14]; pix4[7] <= p2[15]; // 88
540
541        x_r[0] = 5; x_r[1] = 6; x_r[2] = 7; x_r[3] = 8;
542        x_r[4] = 9; x_r[5] = 10; x_r[6] = 11; x_r[7] = 12;
543
544        reg_shift_r <= 1'b0;
545        end
546
547
548        4'b1011: begin
549        pix1[0] <= p1[14]; pix2[0] <= p1[15];
550        pix1[1] <= p1[15]; pix2[1] <= p1[16];
551        pix1[2] <= p1[16]; pix2[2] <= p1[17];
552        pix1[3] <= p1[16]; pix2[3] <= p1[17];
553        pix1[4] <= p1[17]; pix2[4] <= p1[18];
554        pix1[5] <= p1[17]; pix2[5] <= p1[18];
555        pix1[6] <= p1[18]; pix2[6] <= p1[19];
556        pix1[7] <= p1[18]; pix2[7] <= p1[19];
557
558        pix3[0] <= p2[14]; pix4[0] <= p2[15];
559        pix3[1] <= p2[15]; pix4[1] <= p2[16]; // 90
```

```
560         pix3[2] <= p2[16]; pix4[2] <= p2[17];
561         pix3[3] <= p2[16]; pix4[3] <= p2[17];
562         pix3[4] <= p2[17]; pix4[4] <= p2[18];
563         pix3[5] <= p2[17]; pix4[5] <= p2[18];
564         pix3[6] <= p2[18]; pix4[6] <= p2[19];
565         pix3[7] <= p2[18]; pix4[7] <= p2[19]; // 96
566
567         x_r[0] = 13; x_r[1] = 14; x_r[2] = 0; x_r[3] = 1;
568         x_r[4] = 2; x_r[5] = 3; x_r[6] = 4; x_r[7] = 5;
569
570         // SHIFT IN NEW DATA IN [191:128]
571         reg_shift_r <= 1'b1;
572         end
573
574         4'b1100: begin
575         pix1[0] <= p1[19]; pix2[0] <= p1[20];
576         pix1[1] <= p1[19]; pix2[1] <= p1[20];
577         pix1[2] <= p1[20]; pix2[2] <= p1[21];
578         pix1[3] <= p1[20]; pix2[3] <= p1[21];
579         pix1[4] <= p1[21]; pix2[4] <= p1[22];
580         pix1[5] <= p1[21]; pix2[5] <= p1[22];
581         pix1[6] <= p1[22]; pix2[6] <= p1[23];
582         pix1[7] <= p1[22]; pix2[7] <= p1[23];
583
584         pix3[0] <= p2[19]; pix4[0] <= p2[20];
585         pix3[1] <= p2[19]; pix4[1] <= p2[20];
586         pix3[2] <= p2[20]; pix4[2] <= p2[21];
587         pix3[3] <= p2[20]; pix4[3] <= p2[21];
588         pix3[4] <= p2[21]; pix4[4] <= p2[22];
589         pix3[5] <= p2[21]; pix4[5] <= p2[22];
590         pix3[6] <= p2[22]; pix4[6] <= p2[23];
591         pix3[7] <= p2[22]; pix4[7] <= p2[23]; // 104
592
593         x_r[0] = 6; x_r[1] = 7; x_r[2] = 8; x_r[3] = 9;
594         x_r[4] = 10; x_r[5] = 11; x_r[6] = 12; x_r[7] = 13;
595
596         reg_shift_r <= 1'b0;
597         end
598
599         4'b1101: begin
600         pix1[0] <= p1[23]; pix2[0] <= p1[24];
601         pix1[1] <= p1[24]; pix2[1] <= p1[25];
602         pix1[2] <= p1[24]; pix2[2] <= p1[25];
603         pix1[3] <= p1[25]; pix2[3] <= p1[26];
604         pix1[4] <= p1[25]; pix2[4] <= p1[26];
605         pix1[5] <= p1[26]; pix2[5] <= p1[27];
606         pix1[6] <= p1[26]; pix2[6] <= p1[27];
607         pix1[7] <= p1[27]; pix2[7] <= p1[28];
608
609         pix3[0] <= p2[23]; pix4[0] <= p2[24]; // 105
610         pix3[1] <= p2[24]; pix4[1] <= p2[25];
```

```
611          pix3[2] <= p2[24]; pix4[2] <= p2[25];
612          pix3[3] <= p2[25]; pix4[3] <= p2[26];
613          pix3[4] <= p2[25]; pix4[4] <= p2[26];
614          pix3[5] <= p2[26]; pix4[5] <= p2[27];
615          pix3[6] <= p2[26]; pix4[6] <= p2[27];
616          pix3[7] <= p2[27]; pix4[7] <= p2[28]; // 112
617
618          x_r[0] = 14; x_r[1] = 0; x_r[2] = 1; x_r[3] = 2;
619          x_r[4] = 3; x_r[5] = 4; x_r[6] = 5; x_r[7] = 6;
620
621          // SHIFT IN NEW DATA IN [127:64]
622          reg_shift_r <= 1'b1;
623          end
624
625          4'b1110: begin // 14
626          pix1[0] <= p1[27]; pix2[0] <= p1[28];
627          pix1[1] <= p1[28]; pix2[1] <= p1[29];
628          pix1[2] <= p1[28]; pix2[2] <= p1[29];
629          pix1[3] <= p1[29]; pix2[3] <= p1[30];
630          pix1[4] <= p1[29]; pix2[4] <= p1[30];
631          pix1[5] <= p1[30]; pix2[5] <= p1[31];
632          pix1[6] <= p1[30]; pix2[6] <= p1[31];
633          pix1[7] <= p1[31]; pix2[7] <= p1[0];
634
635          pix3[0] <= p2[27]; pix4[0] <= p2[28];
636          pix3[1] <= p2[28]; pix4[1] <= p2[29];
637          pix3[2] <= p2[28]; pix4[2] <= p2[29];
638          pix3[3] <= p2[29]; pix4[3] <= p2[30];
639          pix3[4] <= p2[29]; pix4[4] <= p2[30];
640          pix3[5] <= p2[30]; pix4[5] <= p2[31];
641          pix3[6] <= p2[30]; pix4[6] <= p2[31];
642          pix3[7] <= p2[31]; pix4[7] <= p2[0]; // 120
643
644          x_r[0] = 7; x_r[1] = 8; x_r[2] = 9; x_r[3] = 10;
645          x_r[4] = 11; x_r[5] = 12; x_r[6] = 13; x_r[7] = 14;
646
647          // SHIFT IN NEW DATA IN [63:0]
648          reg_shift_r <= 1'b1;
649          line_count <= 0;
650          end
651
652        endcase
653
654
655        //------------------------ Check horizontal -------------------------//
656        if (horizontal_count == FRAME_WIDTH_OUT-1) begin // A line is finished.
657          horizontal_count <= 0;
658
659          if (y_count == 14)
660            y_count <= 0;
661          else
```

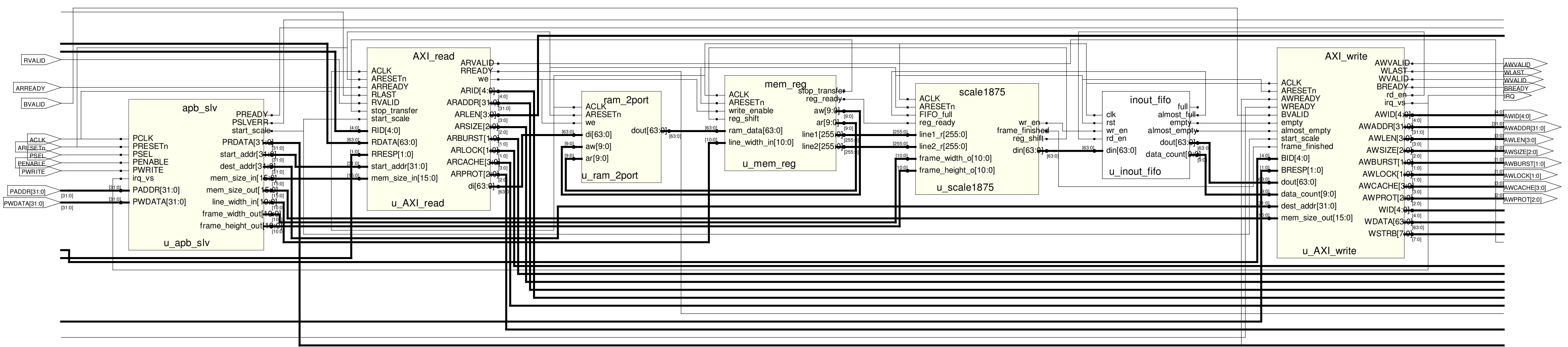
```
662           y_count <= y_count + 1;
663
664         if (line_number == 14) begin
665           line_number <= 0;
666         end
667         else
668           line_number <= line_number + 1;
669
670         //------------------------ Check Vertical -------------------------//
671         if (vertical_count == FRAME_HEIGHT_OUT-1) begin
672           vertical_count <= 0;
673           frame_finished_r <= 1'b1;
674           if (yuv_count == 2)
675             yuv_count <= 0;
676           else
677             yuv_count <= yuv_count + 1;
678         end
679         else
680           vertical_count <= vertical_count + 1;
681
682       end
683       else
684         horizontal_count <= horizontal_count + 1;
685
686
687     end // if (FIFO is not full AND registers are ready)
688
689     if (wr_en_r)
690       dummy_count <= dummy_count + 1;
691
692     if (((horizontal_count == FRAME_WIDTH_OUT-1) ||
693          (horizontal_count == FRAME_WIDTH_OUT-2) ||
694          (horizontal_count == FRAME_WIDTH_OUT-3) ||
695          (horizontal_count == FRAME_WIDTH_OUT-4) ) &&
696          (vertical_count == FRAME_HEIGHT_OUT-1) ) begin
697       if (yuv_count == 2) begin
698         last_data_packet <= 1;
699       end
700     end
701
702   end // elsif(posedge ACLK)
703 end // always
704
705
706 /*****************************************************************************
707 * Drive outputs
708 *****************************************************************************/
709 assign din            = fifo_data;
710 assign wr_en          = wr_en_r;
711 assign frame_finished = last_data_packet; //frame_finished_r
712 assign reg_shift      = reg_shift_r;
```

```
713
714 endmodule
```

B.7 RAM module

```
/******************************************************************************/
/*                                                                            */
/* Author: Roger Skarboe                                                      */
/*                                                                            */
/* Version: v 1.00, 29.04.2010                                                */
/*                                                                            */
/* Description: 2 port RAM, input from AXI Read, output to scale module.     */
/*                                                                            */
/******************************************************************************/
module ram_2port (ACLK, ARESETn, di, aw, we, dout, ar);

/* Define constants for output signals */
parameter di_WIDTH   = 64;
parameter ADDR_WIDTH = 10;
parameter RAM_DEPTH  = 1024;
//parameter RAM_DEPTH = 1 << ADDR_WIDTH;

/* Global input signals */
input                     ACLK; // Clock
input                     ARESETn; // Reset for testing.

/* 2 port RAM signals */
input   [di_WIDTH-1:0]    di; // Data in
input   [ADDR_WIDTH-1:0]  aw; // Write address
input                     we; // Write enable
output  [di_WIDTH-1:0]    dout; // Data out
input   [ADDR_WIDTH-1:0]  ar; // Read address

/******************************************************************************
* Register space
******************************************************************************/
reg     [di_WIDTH-1:0]    ram [0:RAM_DEPTH-1];
reg     [15:0]            i; // For testing


initial begin
  for (i = 0; i < RAM_DEPTH; i = i+1)
    ram[i] <= 0;
end

// Process: Memory Write Block
always @(posedge ACLK) // or negedge ARESETn
begin: write_mem
  if (we)
    ram[aw] <= di;
end

assign dout = ram[ar];
endmodule
```

C Video scaler system


apb_slv
u_apb_slv
AXI_read
u_AXI_read
ram_2port
u_ram_2port
mem_reg
u_mem_reg
scale1875
u_scale1875
inout_fifo
u_inout_fifo
AXI_write
u_AXI_write

D Log files

D.1 Video scaler 1.25 - Xilinx log file

```
Release 11.3 Map L.57 (lin64)
Xilinx Map Application Log File for Design 'LT_XC5VLX330_FPGA'

Design Information
------------------
Command Line   : map -intstyle ise -p xc5vlx330-ff1760-1 -t 1 -c 100 -tx off
-timing -cm area -pr b -ol high -xe c -o fpga_map.ncd fpga.ngd fpga.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.51.18.1 $
Mapped Date    : Wed Jun  2 19:16:37 2010

....

Running timing-driven placement...
Total REAL time at the beginning of Placer: 2 mins 47 secs
Total CPU  time at the beginning of Placer: 2 mins 45 secs

Phase 1.1  Initial Placement Analysis
Phase 1.1  Initial Placement Analysis (Checksum:19760a30) REAL time: 3 mins 9 secs

Phase 2.7  Design Feasibility Check
Phase 2.7  Design Feasibility Check (Checksum:19760a30) REAL time: 3 mins 10 secs

Phase 3.31  Local Placement Optimization
Phase 3.31  Local Placement Optimization (Checksum:19760a30) REAL time: 3 mins 10 secs

Phase 4.37  Local Placement Optimization
Phase 4.37  Local Placement Optimization (Checksum:19760a30) REAL time: 3 mins 10 secs

Phase 5.33  Local Placement Optimization
Phase 5.33  Local Placement Optimization (Checksum:19760a30) REAL time: 5 mins 36 secs

Phase 6.32  Local Placement Optimization
Phase 6.32  Local Placement Optimization (Checksum:19760a30) REAL time: 5 mins 44 secs

Phase 7.2  Initial Clock and IO Placement

Phase 7.2  Initial Clock and IO Placement (Checksum:ebe35775) REAL time: 6 mins 51 secs

Phase 8.36  Local Placement Optimization
Phase 8.36  Local Placement Optimization (Checksum:ebe35775) REAL time: 6 mins 51 secs

Phase 9.30  Global Clock Region Assignment
Phase 9.30  Global Clock Region Assignment (Checksum:ebe35775) REAL time: 6 mins 51 secs

Phase 10.3  Local Placement Optimization
Phase 10.3  Local Placement Optimization (Checksum:ebe35775) REAL time: 6 mins 52 secs

Phase 11.5  Local Placement Optimization
Phase 11.5  Local Placement Optimization (Checksum:ebe35775) REAL time: 6 mins 54 secs

Phase 12.8  Global Placement
.........................................................................
.............................................
...................................................
.........................................................................
```

```
59  Phase 12.8  Global Placement (Checksum:fd741de6) REAL time: 17 mins 11 secs
60
61  Phase 13.29  Local Placement Optimization
62  Phase 13.29  Local Placement Optimization (Checksum:fd741de6) REAL time: 17 mins 11 secs
63
64  Phase 14.5  Local Placement Optimization
65  Phase 14.5  Local Placement Optimization (Checksum:fd741de6) REAL time: 17 mins 16 secs
66
67  Phase 15.18  Placement Optimization
68  Phase 15.18  Placement Optimization (Checksum:c0afacfe) REAL time: 26 mins 45 secs
69
70  Phase 16.5  Local Placement Optimization
71  Phase 16.5  Local Placement Optimization (Checksum:c0afacfe) REAL time: 26 mins 50 secs
72
73  Phase 17.34  Placement Validation
74  Phase 17.34  Placement Validation (Checksum:c0afacfe) REAL time: 26 mins 53 secs
75
76  Total REAL time to Placer completion: 27 mins
77  Total CPU  time to Placer completion: 26 mins 58 secs
78  Running post-placement packing...
79  Writing output files...
80
81  Design Summary
82  --------------
83
84  Design Summary:
85  Number of errors:      0
86  Number of warnings:  203
87  Slice Logic Utilization:
88    Number of Slice Registers:                23,321 out of 207,360   11%
89      Number used as Flip Flops:              23,321
90    Number of Slice LUTs:                     30,974 out of 207,360   14%
91      Number used as logic:                   26,740 out of 207,360   12%
92        Number using O6 output only:          26,018
93        Number using O5 output only:             210
94        Number using O5 and O6:                  512
95      Number used as Memory:                   4,214 out of  54,720    7%
96        Number used as Dual Port RAM:          4,211
97          Number using O5 output only:             5
98          Number using O5 and O6:              4,206
99        Number used as Shift Register:             3
100         Number using O6 output only:             3
101     Number used as exclusive route-thru:        20
102   Number of route-thrus:                       257
103     Number using O6 output only:               228
104     Number using O5 output only:                29
105
106 Slice Logic Distribution:
107   Number of occupied Slices:                14,093 out of  51,840   27%
108   Number of LUT Flip Flop pairs used:       42,758
109     Number with an unused Flip Flop:        19,437 out of  42,758   45%
110     Number with an unused LUT:              11,784 out of  42,758   27%
111     Number of fully used LUT-FF pairs:      11,537 out of  42,758   26%
112     Number of unique control sets:             754
113     Number of slice register sites lost
114       to control set restrictions:             923 out of 207,360    1%
115
116   A LUT Flip Flop pair for this architecture represents one LUT paired with
117   one Flip Flop within a slice.  A control set is a unique combination of
118   clock, reset, set, and enable signals for a registered element.
119   The Slice Logic Distribution report is not meaningful if the design is
120   over-mapped for a non-slice resource or if Placement fails.
121   OVERMAPPING of BRAM resources should be ignored if the design is
122   over-mapped for a non-BRAM resource or if placement fails.
123
```

```
124 IO Utilization:
125   Number of bonded IOBs:                    620 out of   1,200   51%
126     Number of LOCed IOBs:                   620 out of     620  100%
127     IOB Flip Flops:                         601
128
129 Specific Feature Utilization:
130   Number of BlockRAM/FIFO:                   20 out of     288    6%
131     Number using BlockRAM only:              16
132     Number using FIFO only:                   4
133     Total primitives used:
134       Number of 36k BlockRAM used:           13
135       Number of 18k BlockRAM used:            3
136       Number of 36k FIFO used:                4
137     Total Memory used (KB):                 666 out of  10,368    6%
138   Number of BUFG/BUFGCTRLs:                   8 out of      32   25%
139     Number used as BUFGs:                     8
140   Number of DCM_ADVs:                         2 out of      12   16%
141   Number of DSP48Es:                         32 out of     192   16%
142
143 Average Fanout of Non-Clock Nets:             4.07
144
145 Peak Memory Usage:  2059 MB
146 Total REAL time to MAP completion:  28 mins 10 secs
147 Total CPU time to MAP completion:   28 mins 8 secs
148
149 Mapping completed.
150 See MAP report file "fpga_map.mrp" for details.
```

D.2 Video scaler 1.875 - Xilinx log file

```
1  Release 11.3 Map L.57 (lin64)
2  Xilinx Map Application Log File for Design 'LT_XC5VLX330_FPGA'
3
4  Design Information
5  ------------------
6  Command Line   : map -intstyle ise -p xc5vlx330-ff1760-1 -t 1 -c 100 -tx off
7  -timing -cm area -pr b -ol high -xe c -o fpga_map.ncd fpga.ngd fpga.pcf
8  Target Device  : xc5vlx330
9  Target Package : ff1760
10 Target Speed   : -1
11 Mapper Version : virtex5 -- $Revision: 1.51.18.1 $
12 Mapped Date    : Mon Jun  7 12:05:35 2010
13
14 ....
15
16 Running timing-driven placement...
17 Total REAL time at the beginning of Placer: 3 mins
18 Total CPU  time at the beginning of Placer: 2 mins 57 secs
19
20 Phase 1.1  Initial Placement Analysis
21 Phase 1.1  Initial Placement Analysis (Checksum:c1e76f9e) REAL time: 3 mins 24 secs
22
23 Phase 2.7  Design Feasibility Check
24 Phase 2.7  Design Feasibility Check (Checksum:c1e76f9e) REAL time: 3 mins 25 secs
25
26 Phase 3.31  Local Placement Optimization
27 Phase 3.31  Local Placement Optimization (Checksum:c1e76f9e) REAL time: 3 mins 25 secs
28
29 Phase 4.37  Local Placement Optimization
30 Phase 4.37  Local Placement Optimization (Checksum:c1e76f9e) REAL time: 3 mins 25 secs
31
32 Phase 5.33  Local Placement Optimization
33 Phase 5.33  Local Placement Optimization (Checksum:c1e76f9e) REAL time: 6 mins 3 secs
34
35 Phase 6.32  Local Placement Optimization
36 Phase 6.32  Local Placement Optimization (Checksum:c1e76f9e) REAL time: 6 mins 11 secs
37
38 Phase 7.2  Initial Clock and IO Placement
39
40 Phase 7.2  Initial Clock and IO Placement (Checksum:2a955541) REAL time: 7 mins 21 secs
41
42 Phase 8.36  Local Placement Optimization
43 Phase 8.36  Local Placement Optimization (Checksum:2a955541) REAL time: 7 mins 21 secs
44
45 Phase 9.30  Global Clock Region Assignment
46 Phase 9.30  Global Clock Region Assignment (Checksum:2a955541) REAL time: 7 mins 21 secs
47
48 Phase 10.3  Local Placement Optimization
49 Phase 10.3  Local Placement Optimization (Checksum:2a955541) REAL time: 7 mins 22 secs
50
51 Phase 11.5  Local Placement Optimization
52 Phase 11.5  Local Placement Optimization (Checksum:2a955541) REAL time: 7 mins 24 secs
53
54 Phase 12.8  Global Placement
55 ....................................
56 .........................................................
57 ...................................................................
58 Phase 12.8  Global Placement (Checksum:22b69109) REAL time: 17 mins 11 secs
59
60 Phase 13.29  Local Placement Optimization
61 Phase 13.29  Local Placement Optimization (Checksum:22b69109) REAL time: 17 mins 11 secs
62
```

```
63  Phase 14.5  Local Placement Optimization
64  Phase 14.5  Local Placement Optimization (Checksum:22b69109) REAL time: 17 mins 16 secs
65
66  Phase 15.18  Placement Optimization
67  Phase 15.18  Placement Optimization (Checksum:cce083f6) REAL time: 27 mins 28 secs
68
69  Phase 16.5  Local Placement Optimization
70  Phase 16.5  Local Placement Optimization (Checksum:cce083f6) REAL time: 27 mins 33 secs
71
72  Phase 17.34  Placement Validation
73  Phase 17.34  Placement Validation (Checksum:cce083f6) REAL time: 27 mins 36 secs
74
75  Total REAL time to Placer completion: 27 mins 43 secs
76  Total CPU  time to Placer completion: 27 mins 40 secs
77  Running post-placement packing...
78  Writing output files...
79
80  Design Summary
81  --------------
82
83  Design Summary:
84  Number of errors:      0
85  Number of warnings:  199
86  Slice Logic Utilization:
87    Number of Slice Registers:                23,372 out of 207,360   11%
88      Number used as Flip Flops:              23,372
89    Number of Slice LUTs:                     32,640 out of 207,360   15%
90      Number used as logic:                   28,389 out of 207,360   13%
91        Number using O6 output only:          27,411
92        Number using O5 output only:             233
93        Number using O5 and O6:                  745
94      Number used as Memory:                   4,214 out of  54,720    7%
95        Number used as Dual Port RAM:          4,211
96          Number using O5 output only:             5
97          Number using O5 and O6:              4,206
98        Number used as Shift Register:             3
99          Number using O6 output only:             3
100     Number used as exclusive route-thru:        37
101   Number of route-thrus:                       362
102     Number using O6 output only:               269
103     Number using O5 output only:                93
104
105 Slice Logic Distribution:
106   Number of occupied Slices:                14,407 out of  51,840   27%
107   Number of LUT Flip Flop pairs used:       43,962
108     Number with an unused Flip Flop:        20,590 out of  43,962   46%
109     Number with an unused LUT:              11,322 out of  43,962   25%
110     Number of fully used LUT-FF pairs:      12,050 out of  43,962   27%
111     Number of unique control sets:             755
112     Number of slice register sites lost
113       to control set restrictions:             932 out of 207,360    1%
114
115   A LUT Flip Flop pair for this architecture represents one LUT paired with
116   one Flip Flop within a slice.  A control set is a unique combination of
117   clock, reset, set, and enable signals for a registered element.
118   The Slice Logic Distribution report is not meaningful if the design is
119   over-mapped for a non-slice resource or if Placement fails.
120   OVERMAPPING of BRAM resources should be ignored if the design is
121   over-mapped for a non-BRAM resource or if placement fails.
122
123 IO Utilization:
124   Number of bonded IOBs:                       620 out of   1,200   51%
125     Number of LOCed IOBs:                      620 out of     620  100%
126     IOB Flip Flops:                            601
127
```

```
128 Specific Feature Utilization:
129   Number of BlockRAM/FIFO:                    20 out of     288    6%
130     Number using BlockRAM only:               16
131     Number using FIFO only:                    4
132     Total primitives used:
133       Number of 36k BlockRAM used:            13
134       Number of 18k BlockRAM used:             3
135       Number of 36k FIFO used:                 4
136     Total Memory used (KB):                  666 out of  10,368    6%
137   Number of BUFG/BUFGCTRLs:                    8 out of      32   25%
138     Number used as BUFGs:                      8
139   Number of DCM_ADVs:                          2 out of      12   16%
140   Number of DSP48Es:                          48 out of     192   25%
141
142 Average Fanout of Non-Clock Nets:              4.04
143
144 Peak Memory Usage:  2137 MB
145 Total REAL time to MAP completion:  28 mins 56 secs
146 Total CPU time to MAP completion:   28 mins 52 secs
147
148 Mapping completed.
149 See MAP report file "fpga_map.mrp" for details.
```

D.3 Scale 1.25 module - Synplify log file

```
#Build: Synplify Premier C-2009.06-SP1, Build 074R, Aug 18 2009
#install: /arm/tools/synplicity/synplify/2009.06-SP1/fpga_c200906sp1
#OS: Linux
#Hostname: trd-lsf4.trondheim.arm.com

#Implementation: rev_2

#Mon Jun 14 12:50:17 2010

....

Performance Summary
*******************


Worst slack in design: 997.121

                       Requested     Estimated     Requested     Estimated
                           Clock         Clock
Starting Clock         Frequency     Frequency     Period        Period        Slack
    Type           Group
-----------------------------------------------------------------------------------
    --------------------------- ---------------------
scale125|ACLK          1.0 MHz       347.3 MHz     1000.000      2.879         997.121
    inferred       Inferred_clkgroup_0
===================================================================================
    =========================== ======================





Clock Relationships
*******************

Clocks                         |    rise  to  rise    |    fall  to  fall   |    rise  to
     fall   |    fall  to  rise
-----------------------------------------------------------------------------------
    -------------------------- --------------------
Starting        Ending         |  constraint  slack   |  constraint  slack  |  constraint
     slack  |  constraint  slack
-----------------------------------------------------------------------------------
    -------------------------- --------------------
scale125|ACLK   scale125|ACLK  |  1000.000    997.121  |  No paths    -      |  No paths
       -       |  No paths    -
===================================================================================
    ========================== ====================
 Note: 'No paths' indicates there are no paths in the design for that pair of clock edges.
       'Diff grp' indicates that paths exist but the starting clock and ending clock are
           in different clock groups.



Interface Information
*********************

    No IO constraint found

...

##### END OF TIMING REPORT #####]

```

```
51  ---------------------------------------
52  Resource Usage Report for scale125
53
54  Mapping to part: xc5vlx330ff1760-2
55  Cell usage:
56  DSP48E          32 uses
57  FDC             8 uses
58  FDCE            20 uses
59  FDE             266 uses
60  GND             1 use
61  MUXCY           4 uses
62  MUXCY_L         39 uses
63  VCC             1 use
64  XORCY           35 uses
65  LUT1            30 uses
66  LUT2            16 uses
67  LUT3            3 uses
68  LUT4            2 uses
69  LUT5            3 uses
70  LUT6            382 uses
71  LUT6_2            200 uses
72
73  I/O ports: 605
74  I/O primitives: 602
75  IBUF           534 uses
76  IBUFG          1 use
77  OBUF           67 uses
78
79  BUFG           1 use
80
81  I/O Register bits:                  0
82  Register bits not including I/Os:   294 (0%)
83
84  DSP48s: 32 of 192 (16%)
85
86  Global Clock Buffers: 1 of 32 (3%)
87
88
89   Number of unique control sets:              4
90   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_180_i)    : 8
91   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(VCC)    : 8
92   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(wr_en_r6)     : 12
93   C(ACLK_c), CLR(GND), PRE(GND), CE(pix1_0__0_sqmuxa)     : 266
94
95  Total load per clock:
96     scale125|ACLK: 326
97
98  Mapping Summary:
99  Total  LUTs: 636 (0%)
100
101 @N: MF234 |Hierarchical island-based critical path report is located in
        /home/rogska01/master/synplify/scale_125/rev_2/scale125.tah
102
103 Mapper successful!
104 Process took 0h:00m:25s realtime, 0h:00m:25s cputime
105 # Mon Jun 14 12:50:48 2010
106
107 ###########################################################]
```

D.4 Scale 1.875 module - Synplify log file

```
1  #Build: Synplify Premier C-2009.06-SP1, Build 074R, Aug 18 2009
2  #install: /arm/tools/synplicity/synplify/2009.06-SP1/fpga_c200906sp1
3  #OS: Linux
4  #Hostname: trd-lsf4.trondheim.arm.com
5
6  #Implementation: rev_1
7
8  #Mon Jun 14 13:00:02 2010
9
10 ....
11
12 Performance Summary
13 *******************
14
15
16 Worst slack in design: -0.388
17
18                    Requested      Estimated      Requested      Estimated
                          Clock          Clock
19 Starting Clock     Frequency      Frequency      Period         Period         Slack       Type
             Group
20 ------------------------------------------------------------------------------------------------
        ------------------------------------------------------------
21 scale1875|ACLK     396.8 MHz      343.9 MHz      2.520          2.908          -0.388
     inferred      Autoconstr_clkgroup_0
22 ================================================================================================
        ============================================================
23
24
25
26
27
28 Clock Relationships
29 *******************
30
31 Clocks                              |    rise  to  rise    |    fall  to  fall    |    rise  to
       fall    |     fall  to  rise
32 ----------------------------------------------------------------------------------------------------
        ------------------------------------------------------
33 Starting        Ending              |  constraint  slack   |  constraint  slack  |  constraint
       slack  |  constraint  slack
34 ----------------------------------------------------------------------------------------------------
        ------------------------------------------------------
35 scale1875|ACLK  scale1875|ACLK  |  2.520        -0.388  |  No paths    -      |  No paths
       -      |  No paths    -
36 ====================================================================================================
        ======================================================
37  Note: 'No paths' indicates there are no paths in the design for that pair of clock edges.
38        'Diff grp' indicates that paths exist but the starting clock and ending clock are
            in different clock groups.
39
40
41
42 Interface Information
43 *********************
44
45     No IO constraint found
46
47 ....
48
49 ##### END OF TIMING REPORT #####]
50
```

```
51 ---------------------------------------------------------------
52  Resource Usage Report for scale1875
53
54  Mapping to part: xc5vlx330ff1760-1
55  Cell usage:
56  DSP48E          64 uses
57  FDC             9 uses
58  FDCE            166 uses
59  FDE             480 uses
60  GND             1 use
61  MUXCY           10 uses
62  MUXCY_L         79 uses
63  VCC             1 use
64  XORCY           68 uses
65  LUT1            48 uses
66  LUT2            30 uses
67  LUT3            9 uses
68  LUT4            172 uses
69  LUT5            303 uses
70  LUT6            909 uses
71  LUT6_2            53 uses
72
73  I/O ports: 605
74  I/O primitives: 602
75  IBUF           534 uses
76  IBUFG          1 use
77  OBUF           67 uses
78
79  BUFG           1 use
80
81  I/O Register bits:                  0
82  Register bits not including I/Os:   655 (0%)
83
84  DSP48s: 64 of 192 (33%)
85
86  Global Clock Buffers: 1 of 32 (3%)
87
88
89   Number of unique control sets:              4
90   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(wr_en_r6_i)     : 155
91   C(ACLK_c), CLR(GND), PRE(GND), CE(line_count_fast_fast_fast_RNI8CV3[2])      : 480
92   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_25_i_1)     : 11
93   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(VCC)    : 9
94
95  Total load per clock:
96     scale1875|ACLK: 679
97
98  Mapping Summary:
99  Total  LUTs: 1524 (0%)
100
101 @N: MF234 |Hierarchical island-based critical path report is located in
        /home/rogska01/master/synplify/scale_1875/rev_1/scale1875.tah
102
103 Mapper successful!
104 Process took 0h:01m:43s realtime, 0h:01m:42s cputime
105 # Mon Jun 14 13:01:52 2010
106
107 ###########################################################
```

D.5 Mem_reg module - Synplify log file

```
1  #Build: Synplify Premier C-2009.06-SP1, Build 074R, Aug 18 2009
2  #install: /arm/tools/synplicity/synplify/2009.06-SP1/fpga_c200906sp1
3  #OS: Linux
4  #Hostname: trd-lsf4.trondheim.arm.com
5
6  #Implementation: rev_1
7
8  #Mon Jun 14 13:07:20 2010
9
10 ....
11
12 Performance Summary
13 *******************
14
15
16 Worst slack in design: -0.752
17
18                      Requested      Estimated      Requested      Estimated
                           Clock          Clock
19 Starting Clock       Frequency      Frequency      Period         Period         Slack       Type
              Group
20 ------------------------------------------------------------------------------------------
      ------------------------------------------------------
21 mem_reg|ACLK         234.8 MHz      199.6 MHz      4.259          5.011          -0.752
      inferred      Autoconstr_clkgroup_0
22 ==========================================================================================
      ======================================================
23
24
25
26
27
28 Clock Relationships
29 *******************
30
31 Clocks                        |     rise  to  rise     |     fall  to  fall    |     rise  to
      fall    |     fall  to  rise
32 ------------------------------------------------------------------------------------------
      ------------------------------------------------
33 Starting       Ending         |  constraint  slack    |  constraint  slack  |  constraint
      slack  |  constraint  slack
34 ------------------------------------------------------------------------------------------
      ------------------------------------------------
35 mem_reg|ACLK   mem_reg|ACLK   |  4.259       -0.752   |  No paths    -      |  No paths    -
         |  No paths    -
36 ==========================================================================================
      ================================================
37  Note: 'No paths' indicates there are no paths in the design for that pair of clock edges.
38        'Diff grp' indicates that paths exist but the starting clock and ending clock are
          in different clock groups.
39
40
41
42 Interface Information
43 *********************
44
45     No IO constraint found
46
47 ...
48
49 ##### END OF TIMING REPORT #####]
50
```

```
51 ---------------------------------------------------------------
52  Resource Usage Report for mem_reg
53
54  Mapping to part: xc5vlx330ff1760-1
55  Cell usage:
56  FDC             54 uses
57  FDCE            520 uses
58  FDP             3 uses
59  GND             1 use
60  MUXCY           9 uses
61  MUXCY_L         46 uses
62  VCC             1 use
63  XORCY           37 uses
64  LUT1            28 uses
65  LUT2            8 uses
66  LUT3            17 uses
67  LUT4            26 uses
68  LUT5            29 uses
69  LUT6            57 uses
70  LUT6_2            7 uses
71
72  I/O ports: 614
73  I/O primitives: 611
74  IBUF           76 uses
75  IBUFG          1 use
76  OBUF           534 uses
77
78  BUFG           1 use
79
80  I/O Register bits:                     0
81  Register bits not including I/Os:    577 (0%)
82
83  Global Clock Buffers: 1 of 32 (3%)
84
85
86   Number of unique control sets:               11
87   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(VCC)    : 54
88   C(ACLK_c), CLR(GND), PRE(ARESETn_c_i), CE(VCC)    : 3
89   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_17)     : 8
90   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(l2_0__1_sqmuxa)     : 64
91   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(l2_1__1_sqmuxa)     : 64
92   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(l2_2__1_sqmuxa)     : 64
93   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(reg_pos_1_sqmuxa)     : 64
94   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_4)    : 64
95   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_6)    : 64
96   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_8)    : 64
97   C(ACLK_c), CLR(ARESETn_c_i), PRE(GND), CE(N_10)     : 64
98
99  Total load per clock:
100    mem_reg|ACLK: 577
101
102 Mapping Summary:
103 Total  LUTs: 172 (0%)
104
105 @N: MF234 |Hierarchical island-based critical path report is located in
        /home/rogska01/master/synplify/mem_reg1875/rev_1/mem_reg1875.tah
106
107 Mapper successful!
108 Process took 0h:00m:28s realtime, 0h:00m:28s cputime
109 # Mon Jun 14 13:07:52 2010
110
111 ###########################################################
```