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
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 upsampling 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  

III
8 Discussion
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

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bilinear interpolation, [2].</td>
</tr>
<tr>
<td>2</td>
<td>Scale2x algorithm. Shows expansion of one input pixel.</td>
</tr>
<tr>
<td>3</td>
<td>Original picture (zoom 200 %) and Scale2x effect.</td>
</tr>
<tr>
<td>4</td>
<td>Scale3x algorithm. Shows expansion of one input pixel.</td>
</tr>
<tr>
<td>5</td>
<td>Original picture (zoom 300 %) and Scale3x effect.</td>
</tr>
<tr>
<td>6</td>
<td>Original picture (zoom 300 %) and hq3x effect.</td>
</tr>
<tr>
<td>7</td>
<td>Illustration colour distortion of Rogers2x algorithm. Original picture (left, zoom 200 %) and Rogers2x algorithm (right).</td>
</tr>
<tr>
<td>8</td>
<td>Rogers2x algorithm. Shows expansion of one input pixel.</td>
</tr>
<tr>
<td>9</td>
<td>Original picture (zoom 200 %), Scale2x algorithm, and Rogers2x algorithm.</td>
</tr>
<tr>
<td>10</td>
<td>Output dependencies with linear scaling, scaling factor 1.25.</td>
</tr>
<tr>
<td>11</td>
<td>Output dependencies with linear scaling, scaling factor 1.5.</td>
</tr>
<tr>
<td>12</td>
<td>Output dependencies with linear scaling, scaling factor 1.875.</td>
</tr>
<tr>
<td>13</td>
<td>AXI read overlapping burst example (burst of 8 transfers).</td>
</tr>
<tr>
<td>14</td>
<td>AXI write burst example (burst of 8 transfers).</td>
</tr>
<tr>
<td>15</td>
<td>Verilog toplevel testbench structure.</td>
</tr>
<tr>
<td>16</td>
<td>Test of AXI and APB interface.</td>
</tr>
<tr>
<td>17</td>
<td>Memory management: Reading from RAM once.</td>
</tr>
<tr>
<td>18</td>
<td>Output of scaling factor 1.25 with one RAM read.</td>
</tr>
<tr>
<td>19</td>
<td>Output of scaling factor 1.875 with one RAM read.</td>
</tr>
<tr>
<td>20</td>
<td>Memory management: Reading from RAM twice.</td>
</tr>
<tr>
<td>21</td>
<td>Internal register representation of Scale 1.25 module and output.</td>
</tr>
<tr>
<td>22</td>
<td>Internal register representation of Scale 1.875 module and output.</td>
</tr>
<tr>
<td>23</td>
<td>Video scaler system.</td>
</tr>
<tr>
<td>24</td>
<td>Overview of the system implemented on the FPGA.</td>
</tr>
<tr>
<td>25</td>
<td>SD2FullHD accurate to dependencies (400% zoom).</td>
</tr>
<tr>
<td>26</td>
<td>SD2FullHD with smoothing effect (400% zoom).</td>
</tr>
<tr>
<td>27</td>
<td>GIMP Linear output (400% zoom).</td>
</tr>
<tr>
<td>28</td>
<td>Original picture of Lenna’s eye (400% zoom).</td>
</tr>
<tr>
<td>29</td>
<td>Part of the ASCII characters, source picture.</td>
</tr>
<tr>
<td>30</td>
<td>ASCII characters scaled up with a factor of 1.875, bilinear interpolation (top) and SD2FullHD (bottom).</td>
</tr>
<tr>
<td>31</td>
<td>ASCII characters scaled up with a factor of 1.5, bilinear interpolation (top) and SD2FullHD (bottom).</td>
</tr>
<tr>
<td>32</td>
<td>ASCII characters scaled up with a factor of 1.25, bilinear interpolation (top) and SD2FullHD (bottom).</td>
</tr>
</tbody>
</table>
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 YCbCr. 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\textsubscript{b}C\textsubscript{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\textsubscript{b}C\textsubscript{r} and RGB is that YC\textsubscript{b}C\textsubscript{r} represent colour as brightness, Y (luma), and two colour difference signals, C\textsubscript{b} (B-Y) and C\textsubscript{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:

\[
Y = 0.299R + 0.587G + 0.114B \\
C_r = V' = (R-Y)*0.713 \\
C_b = U' = (B-Y)*0.565
\]

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

\[
R = Y + 1.403V' \\
G = Y - 0.344U' - 0.714V' \\
B = Y + 1.770U'
\]

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.

![Bilinear interpolation diagram](image)

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.

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

where $R_1 = (x, y_1)$.

$$f(R_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$ (2)

where $R_2 = (x, y_2)$.

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

$$f(P) = \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2)$$ (3)
Equation 4 shows the full extent of function $f(x, y)$.

$$f(x, y) = \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y)$$
$$+ \frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y_2 - y)$$
$$+ \frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y - y_1)$$
$$+ \frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y - y_1)$$

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

$$f(x, y) = f(0, 0) (1 - x)(1 - y) + f(1, 0) x(1 - y) + f(0, 1) (1 - x)y + f(1, 1)$$  (5)

It is well known 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 $E_0$, $E_1$, $E_2$, and $E_3$. 
Scale2x use a set of rules, these are (described in C):

\[
\begin{align*}
E_0 &= D == B && B != F && D != H ? D : E; \\
E_1 &= B == F && B != D && F != H ? F : E; \\
E_2 &= D == H && D != B && H != F ? D : E; \\
E_3 &= H == F && D != H && B != F ? F : E;
\end{align*}
\]

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

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

\begin{verbatim}
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;
\end{verbatim}

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 into 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\textsubscript{b}C\textsubscript{r}) pixel values, the dependencies on the human eye are considered. The interpolation is then straightforward 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:
\[ Y = 0.299R + 0.587G + 0.114B \]
\[ Cr = V' = (R-Y)\times0.713 \]
\[ Cb = U' = (B-Y)\times0.565 \]

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:
\[ Y = R/4 + G/2 + B/4 \]
\[ Cr = V' = (R-Y)/2 \]
\[ Cb = U' = (B-Y)/2 \]

The same was done for YUV to RGB conversion. [7] uses the following formulas to do YUV to RGB conversion:
\[ R = Y + 1.403V' \]
\[ G = Y - 0.344U' - 0.714V' \]
\[ B = Y + 1.770U' \]

These formulas were introduced to replace YUV to RGB conversion:
\[ R = Y + 2V' \]
\[ G = Y - U'/2 - V'/2 \]
\[ B = Y + 2U' \]

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 $E_0$, 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.](image)

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.](image)

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 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.

![Output dependencies with linear scaling, scaling factor 1.25.](image)

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.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{output_dependencies_linear_scaling}
\caption{Output dependencies with linear scaling, scaling factor 1.5.}
\end{figure}

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
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: $p_1, p_2, p_3, p_4$ to be the input pixels that $p_o$ depends on.

Require: $x$ and $y$ to be modulo of 15 to the real X and Y coordinates of the output frame.

for all $p_o$ (output pixels) do
    if $x$ and $y$ are even then
        $p_o \leftarrow p_1$
    else if $y$ is even then
        $p_o \leftarrow (p_1 * (8 - ((x + 1)/2)) + p_2 * ((x + 1)/2))/8$
    else if $x$ is even then
        $p_o \leftarrow (p_1 * (8 - ((x + 1)/2)) + p_3 * ((x + 1)/2))/8$
    else if $x$ and $y$ are odd
        $p_o \leftarrow (((x + 1)/2) * ((y + 1)/2) * p_4$
        $\quad + ((y + 1) * 4 - ((x + 1)/2) * ((y + 1)/2)) * p_3$
        $\quad + ((x + 1) * 4 - ((x + 1)/2) * ((y + 1)/2)) * p_2$
        $\quad + (8 - ((x + 1)/2)) * (8 - ((y + 1)/2)) * p_1) / 64$
    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 a 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:

```verilog
defmodule 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 PRESETr;
// APB Slave interface
input [31:0] PADDR;
input PSEl;
input PENABLE;
input PWDATA;
output [31:0] PRDATA;
output PREADY;
output PS LIVERR;
// 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 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 correctly. Below it is shown what addresses that assign what registers:

```
'define APB_START_ADDR 12'h00 // Physical start address
           // output [31:0] start_addr
'define APB_DEST_ADDR 12'h04 // Physical destination address
           // output [31:0] dest_addr
'define APB_MEM_IN_SIZE 12'h08 // Size to read
           // output [15:0] mem_size_in
'define APB_MEM_OUT_SIZE 12'h0c // Size to write
           // output [15:0] mem_size_out
'define APB_START_SCALE 12'h10 // To start scaling
           // output start_scale
'define APB_FRAME_W_IN 12'h14 // Input frame width
           // output [10:0] frame_width_in
'define APB_FRAME_W_OUT 12'h18 // Output frame width
           // output [10:0] frame_width_out
'define APB_FRAME_H_OUT 12'h1c // Output frame height
           // output [10:0] frame_height_out
'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>
<thead>
<tr>
<th>SIGNALS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_addr [31:0]</td>
<td>Physical start address. Connected to the AXI Read module. Sets the first memory location to read from when \texttt{start_scale} is asserted.</td>
</tr>
<tr>
<td>dest_addr [31:0]</td>
<td>Physical destination address. Connected to the AXI Write module. Sets the first memory location to write to when \texttt{start_scale} is asserted.</td>
</tr>
<tr>
<td>mem_size_in [15:0]</td>
<td>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.</td>
</tr>
<tr>
<td>mem_size_out [15:0]</td>
<td>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.</td>
</tr>
<tr>
<td>start_scale</td>
<td>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 \texttt{start_scale}.</td>
</tr>
<tr>
<td>frame_width_in [10:0]</td>
<td>Frame width of input. Connected to mem_reg module.</td>
</tr>
<tr>
<td>frame_width_out [10:0]</td>
<td>Frame width of output. Connected to scale module.</td>
</tr>
<tr>
<td>frame_height_out [10:0]</td>
<td>Frame height of output. Connected to scale module.</td>
</tr>
<tr>
<td>irq_vs</td>
<td>Interrupt. Input from the AXI Write module. When irq_vs is HIGH the frame is finished scaled and written to memory.</td>
</tr>
</tbody>
</table>
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.
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.

```verilog
// 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.

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.](image)

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.](image-url)

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 an 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.

```verilog
// 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;
    end

end
```
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:

```verilog
// 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.

---

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.](image)

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
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.
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 parallel, 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.
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.

```verilog
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[5] <= a16; s2[5] <= a00; s3[5] <= a00; s4[5] <= 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.

```verilog
// 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]

end
```

32
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.

```verilog
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;
```
FPGA Implementation of a Video Scaler

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;
end
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
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.

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

Figure 23: Video scaler system.
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.](image)

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 \texttt{tbUtil} 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, \texttt{config.txt}. This \texttt{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 \texttt{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, respectively. 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.

\[
\text{Throughput frames (format: yuv444p)} = \frac{50 \text{ MHz}}{\text{output resolution} \times 3} \tag{6}
\]

\[
\text{Throughput frames (format: yuv420p)} = \frac{50 \text{ MHz}}{\text{output resolution} \times 1.5} \tag{7}
\]

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

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.
### 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

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

Table 3: Video scaler performance in FPGA with scaling factor 1.875.
window filter [3]. GIMP none uses no interpolation, pixels are simply enlarged or removed, as they are when zooming.

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

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.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Corel bicubic</th>
<th>Corel bilinear</th>
<th>SD2-FullHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>325.919</td>
<td>408.102</td>
<td>346.596</td>
</tr>
<tr>
<td>MSE</td>
<td>3.93819</td>
<td>6.19559</td>
<td>4.74291</td>
</tr>
<tr>
<td>PSNR</td>
<td>42.2118</td>
<td>40.2439</td>
<td>41.4043</td>
</tr>
<tr>
<td>RMSE</td>
<td>508.025</td>
<td>637.203</td>
<td>557.518</td>
</tr>
<tr>
<td>AE</td>
<td>871883</td>
<td>890325</td>
<td>880853</td>
</tr>
<tr>
<td>AE (1%)</td>
<td>110285</td>
<td>172068</td>
<td>109630</td>
</tr>
</tbody>
</table>

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).

$$\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Metric} & \text{GIMP} & \text{GIMP} & \text{GIMP} & \text{GIMP} & \text{SD2HD} \\
\text{} & \text{none} & \text{Linear} & \text{Sinc} & \text{Cubic} & \text{} \\
\hline
\text{MAE} & 549.951 & 473.084 & 366.110 & 453.644 & 412.132 \\
\hline
\text{MSE} & 15.4370 & 7.41521 & 3.90219 & 6.13477 & 6.16828 \\
\hline
\text{PSNR} & 36.2791 & 39.4635 & 42.2516 & 40.2868 & 40.2631 \\
\hline
\text{RMSE} & 1005.81 & 697.105 & 505.698 & 634.068 & 635.797 \\
\hline
\text{AE} & 397492 & 404539 & 403169 & 405227 & 398642 \\
\hline
\text{AE (1%)} & 105597 & 92842 & 50112 & 96454 & 71283 \\
\hline
\end{array}$$

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).

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

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.

```verilog
-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, $1/64 p_1$, $7/64 p_2$, $7/64 p_3$ and $49/64 p_4$, the dependence of input pixel $p_1$ 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 $4/64 p_1$, $10/64 p_2$, $10/64 p_3$ and $30/64 p_4$. 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 %.
Table 8: Effect of lacking smoothness.

<table>
<thead>
<tr>
<th>Metric</th>
<th>GIMP Linear</th>
<th>SD2-FullHD</th>
<th>SD2-FullHD*</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>387.021</td>
<td>346.596</td>
<td>339.835</td>
</tr>
<tr>
<td>MSE</td>
<td>4.65961</td>
<td>4.74291</td>
<td>4.45018</td>
</tr>
<tr>
<td>PSNR</td>
<td>41.4812</td>
<td>41.4043</td>
<td>41.6810</td>
</tr>
<tr>
<td>RMSE</td>
<td>552.601</td>
<td>557.518</td>
<td>540.039</td>
</tr>
<tr>
<td>AE</td>
<td>905683</td>
<td>880853</td>
<td>880883</td>
</tr>
<tr>
<td>AE (1%)</td>
<td>130011</td>
<td>109630</td>
<td>105740</td>
</tr>
</tbody>
</table>

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:

\[
\text{if } x \text{ and } y \text{ is odd then} \\
\quad p_o \leftarrow (p_1 \times (8 - ((x + 1)/2)) \times (8 - ((y + 1)/2)) \\
\quad \quad + p_2 \times ((x + 1) \times 4 - ((x + 1)/2) \times ((y + 1)/2)) \\
\quad \quad + p_3 \times ((y + 1) \times 4 - ((x + 1)/2) \times ((y + 1)/2)) \\
\quad \quad + p_4 \times ((x + 1)/2) \times ((y + 1)/2))/64; \\
\text{end if}
\]

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

\[
\text{if } x \text{ and } y \text{ is odd then} \\
\quad p_o \leftarrow (p_1 \times (8 - x') \times (8 - y') \\
\quad \quad + p_2 \times x'(8 - y') \\
\quad \quad + p_3 \times y'(8 - x') \\
\quad \quad + p_4 \times (x' \times y')/64; \\
\text{end if}
\]

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

\[
f(x, y) = f(0, 0) \times (1 - x) \times (1 - y) + f(1, 0) \times (1 - y) + f(0, 1) \times (1 - x) \times y + f(1, 1) \times x \times y
\]

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.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>SOH</td>
</tr>
</tbody>
</table>

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.

![ASCII characters scaled up with a factor of 1.875, bilinear interpolation (top) and SD2FullHD (bottom).](image)

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.

![ASCII characters scaled up with a factor of 1.5, bilinear interpolation (top) and SD2FullHD (bottom).](image)

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.
8.4 Truncation on the algorithms

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

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

\[
\text{if } x \text{ and } y \text{ is odd then}
\]

\[
po \leftarrow (p1 \times (8 - ((x + 1)/2)) \times (8 - ((y + 1)/2))
+ p2 \times ((x + 1) \times 4 - ((x + 1)/2) \times ((y + 1)/2))
+ p3 \times ((y + 1) \times 4 - ((x + 1)/2) \times ((y + 1)/2))
+ p4 \times ((x + 1)/2) \times ((y + 1)/2))/64;
\]

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

Table 9: Effect of truncation versus rounding up.

Table 10 shows the performance of Scale 1.25 module.

Table 10: Scale 1.25 module performance.

Table 11 shows the performance of Scale 1.875 module.

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.
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 into 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 too 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 an 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


    functionalverification/pages/vcs.aspx.

A C++ code

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

```cpp
#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);

YUV funcModulo2(YUV In1, YUV In2, int x);
YUV funcOdd(YUV In0, YUV In1, YUV In2, YUV In3, int x, int y);
YUV avg4_12(YUV In1, YUV In2);
YUV avg8_8(YUV In1, YUV In2);
YUV avg1_3_3_9(YUV In1, YUV In2, YUV In3, YUV In4);
YUV avg2_2_6_6(YUV In1, YUV In2, YUV In3, YUV In4);
YUV avg1_3_3_9(YUV In1, YUV In2, YUV In3, YUV In4);

void SD2HD(BMP ImageIn, BMP &ImageOut) {
    //---------SD--------// //--------------HD-------------//
    // p1 | p2 | p3 | p4 // // po1 | po2 | po3 | po4 | po5 //
    // p5 | p6 | p7 | p8 // --- -> // po6 | po7 | po8 | po9 | po10//
    // p13| p14| p15| p16// // po16| po17| po18| po19| po20//
    //-------------------// // po21| po22| po23| po24| po25//
    //-----------------------------//

    int NewWidth = (int)(ImageIn.TellWidth() * 1.25);
    int NewHeight = (int)(ImageIn.TellHeight() * 1.25);
    int xo, yo = 0;

    ImageOut.SetSize( NewWidth, NewHeight );

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

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

void HD2FullHD(BMP ImageIn, BMP &ImageOut)
{
    //----HD---// //------FullHD-----//
    // p1 | p2 // x1.5 // po1 | po2 | po3 //
    // p3 | p4 // ---> // po4 | po5 | po6 //
    //---------// // po7 | po8 | po9 //
    //-----------------//

    int NewWidth = (int) (ImageIn.TellWidth() * 1.5);
    int NewHeight = (int) (ImageIn.TellHeight() * 1.5);

    int xo, yo = 0;

    ImageOut.SetSize( NewWidth, NewHeight );

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

    for (int j=0 ; j < ImageIn.TellHeight(); j=j+2)
    {
        for (int i=0 ; i < ImageIn.TellWidth(); i=i+2)
        {
            // Safety if picture is not a factor of 2
            if (2 <= ImageIn.TellWidth()-i &&
                2 <= ImageIn.TellHeight()-j)
            {
                YUV p[5], po[10];

                for (int x=1; x<5; x++)
                {
                    p[x] = RGB2YUV(*ImageIn(i+(x-1)%2,j+(x-1)/2));
                }

                // Calculate average values (See avg-functions)
                po[2] = avg8_8(p[1],p[2]);

                //----HD---// //------FullHD-----//
po[4] = avg8_8(p[1],p[3]);
po[8] = avg8_8(p[3],p[4]);
po[5] = avg4_4_4_4(p[1],p[2],p[3],p[4]);
RGB RGBpo[10];
for (int x=1; x < 10; x++)
{
    RGBpo[x] = YUV2RGB(po[x]);
}

xo = (int)i*1.5; // Scale factor 1.5
yo = (int)j*1.5;

for (int x=0; x<3; x++)
{
    for (int y = 0; y<3; y++)
    {
        ImageOut(xo+x,yo+y)->Red = (char) RGBpo[x+y*3+1].R;
        ImageOut(xo+x,yo+y)->Green = (char) RGBpo[x+y*3+1].G;
        ImageOut(xo+x,yo+y)->Blue = (char) RGBpo[x+y*3+1].B;
    }
}

cout << j << "/" << ImageIn.TellHeight() << endl;
}

void SD2FullHD(BMP ImageIn, BMP &ImageOut)
{
    //----SD---//  //FullHD--//
    // p1 | p2 // x1.875 // po // with regards of 4 input pixels
    // p3 | p4 // ----> //--------//
    //--------//

    /* Algorithm SD2FullHD (8x8 -> 15x15)
    * for all po (x and y of input picture)
    * estimate average of all 4 surrounding pixels by using the
    * relationship:
    * po = p1*(16-x-y)/16 + p2*(x-x/16*y/16) + p3*(x-x/16*y/16)
    * + p4*x/16*y/16;
    * when y = 4 and x = 4 symmetri is reached
    * *
    */

    int NewWidth = (int) (ImageIn.TellWidth() * 1.875);
    int NewHeight = (int) (ImageIn.TellHeight() * 1.875);
YUV p[4], pout, p0;
p0.Y = 0; p0.U = 0; p0.V = 0;

ImageOut.SetSize( NewWidth, NewHeight );

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

for (int j=0 ; j < ImageOut.TellHeight()/15; j++)
{
    for (int i=0 ; i < ImageOut.TellWidth()/15; i++)
    {
        for (int y=0; y<15; y++)
        {
            for (int x=0; x<15; x++)
            {
                p[0] = RGB2YUV(*ImageIn(i*8+x/2,j*8+y/2));
                p[1] = RGB2YUV(*ImageIn(i*8+(x/2)+1,j*8+y/2));
                p[2] = RGB2YUV(*ImageIn(i*8+x/2,j*8+(y/2)+1));
                p[3] = RGB2YUV(*ImageIn(i*8+(x/2)+1,j*8+(y/2)+1));

                if ( (x%2)==0 && (y%2)==0) pout = p[0];
                else if (y%2==0) pout = funcModulo2(p[0],p[1],x);
                else if (x%2==0) pout = funcModulo2(p[0],p[2],y);
                else pout = funcOdd(p[0], p[1], p[2], p[3], x, y);

                RGB RGBpout = YUV2RGB(pout);

                ImageOut(i*15+x,j*15+y)->Red = (char) RGBpout.R;
                ImageOut(i*15+x,j*15+y)->Green = (char) RGBpout.G;
                ImageOut(i*15+x,j*15+y)->Blue = (char) RGBpout.B;
            }
        }
        cout << j << " / " << ImageOut.TellHeight()/15 << endl;
    }
}

YUV funcModulo2(YUV In1, YUV In2, int x)
{
    YUV Out;

    Out.Y = (8-((x+1)/2))*In1.Y*0.125 + ((x+1)/2)*In2.Y*0.125;
    Out.U = (8-((x+1)/2))*In1.U*0.125 + ((x+1)/2)*In2.U*0.125;
Out.V = (8-((x+1)/2))*In1.V*0.125 + ((x+1)/2)*In2.V*0.125;

return Out;
}

YUV funcOdd(YUV In0, YUV In1, YUV In2, YUV In3, int x, int y) {
    YUV Out;
    Out.Y = ( ((x+1)/2)*((y+1)/2)*In3.Y +
              ((y+1)*4 - ((x+1)/2)*((y+1)/2))*In2.Y +
              ((x+1)*4 - ((x+1)/2)*((y+1)/2))*In1.Y +
              (8-((x+1)/2))*(8-((y+1)/2))*In0.Y )/64;
    Out.U = ( ((x+1)/2)*((y+1)/2)*In3.U +
              ((y+1)*4 - ((x+1)/2)*((y+1)/2))*In2.U +
              ((x+1)*4 - ((x+1)/2)*((y+1)/2))*In1.U +
              (8-((x+1)/2))*(8-((y+1)/2))*In0.U )/64;
    Out.V = ( ((x+1)/2)*((y+1)/2)*In3.V +
              ((y+1)*4 - ((x+1)/2)*((y+1)/2))*In2.V +
              ((x+1)*4 - ((x+1)/2)*((y+1)/2))*In1.V +
              (8-((x+1)/2))*(8-((y+1)/2))*In0.V )/64;

return Out;
}

YUV avg4_12(YUV In1, YUV In2) {
    YUV Out;
    Out.Y = (4*In1.Y + 12*In2.Y)/16;
    Out.U = (4*In1.U + 12*In2.U)/16;

return Out;
}

YUV avg8_8(YUV In1, YUV In2) {
    YUV Out;
    /*
    Out.Y = (8*In1.Y + 8*In2.Y)/16;
    Out.U = (8*In1.U + 8*In2.U)/16;
    Out.V = (8*In1.V + 8*In2.V)/16;
    */
    Out.Y = (In1.Y + In2.Y)/2;
    Out.U = (In1.U + In2.U)/2;
    Out.V = (In1.V + In2.V)/2;

return Out;
}

YUV avg1_3_3_9(YUV In1, YUV In2, YUV In3, YUV In4) {

}
YUV Out;

return Out;

YUV avg2_2_6_6(YUV In1, YUV In2, YUV In3, YUV In4)
{
    YUV Out;

    return Out;
}

YUV avg4_4_4_4(YUV In1, YUV In2, YUV In3, YUV In4)
{
    YUV Out;
    /*
    */

    return Out;
}

YUV RGB2YUV(RGBApixel In)
{
    YUV Out;
    /* Factors for implement in HW */
    // Out.U = (In.Blue - Out.Y)/2;
    // Out.V = (In.Red - Out.Y)/2;
    /* Original transform factors */
    Out.Y = 0.299*In.Red + 0.587*In.Green + 0.114*In.Blue;
    Out.U = (In.Blue - Out.Y)*0.565;
    Out.V = (In.Red - Out.Y)*0.713;

    return Out;
}

RGB YUV2RGB(YUV In)
{
RGB Out;

/* Factors for implementation in HW */
// Out.R = 2*In.V + In.Y;
// Out.G = In.Y - In.U/2 - In.V/2;

/* Original transform factors */
Out.R = 1.403*In.V + In.Y;
Out.G = In.Y - 0.344*In.U - 0.714*In.V;
Out.B = In.Y + 1.77*In.U;

return Out;

}
A.2 Algorithms for graphic scaling (RGB)

```c
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 ;
            }
        }
    }
}
```

65
ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
ImageOut(i*2+1,j*2)->Red = ImageIn(i,j-1)->Red ==
ImageIn(i+1,j)->Red ?
ImageIn(i,j)->Red;
ImageOut(i*2,j*2+1)->Red = ImageIn(i-1,j)->Red ==
ImageIn(i,j+1)->Red ?
ImageIn(i,j)->Red;
ImageOut(i*2+1,j*2+1)->Red = ImageIn(i,j+1)->Red ==
ImageIn(i+1,j)->Red ?
ImageIn(i,j)->Red;

} else
{
ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;
ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;
ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;
ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;
}

if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
&& j != ImageIn.TellHeight()-1
&& ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
&& ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
&& ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red
&& ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
&& 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)->Green = ImageIn(i-1,j)->Green ==
ImageIn(i,j-1)->Green ?
ImageIn(i,j)->Green;
ImageOut(i*2+1,j*2)->Green = ImageIn(i,j-1)->Green ==
ImageIn(i+1,j)->Green ?
ImageIn(i,j)->Green;
ImageOut(i*2,j*2+1)->Green = ImageIn(i-1,j)->Green ==
ImageIn(i,j+1)->Green ?
ImageIn(i,j)->Green;
ImageOut(i*2+1,j*2+1)->Green = ImageIn(i,j+1)->Green ==
ImageIn(i+1,j)->Green ?
ImageIn(i,j)->Green;
}
else
{
ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
}

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)->Blue != ImageIn(i,j+1)->Blue
&& ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue )
void scale3xAlgorithm(BMP ImageIn, BMP &ImageOut)
{
    int NewWidth = (int) ( ImageIn.TellWidth() * 3);
    int NewHeight = (int) ( ImageIn.TellHeight() * 3);
    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

                {
                    ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
                    ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;
                }
            }
        }
    }
}
&& 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*3,j*3)->Red = ImageIn(i-1,j)->Red ==
    ImageIn(i,j-1)->Red ?
    ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
    ImageOut(i*3+1,j*3)->Red =
    (ImageIn(i-1,j)->Red == ImageIn(i,j-1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i+1,j-1)->Red) ||
    (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i-1,j+1)->Red) ;
    ImageOut(i*3+2,j*3)->Red = ImageIn(i,j+1)->Red ==
    ImageIn(i-1,j)->Red ?
    ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
    ImageOut(i*3,j*3+1)->Red = ImageIn(i-1,j)->Red ==
    ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i+1,j+1)->Red) ||
    (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i,j+1)->Red) ;
    ImageOut(i*3+1,j*3+1)->Red = ImageIn(i,j)->Red;
    ImageOut(i*3+2,j*3+1)->Red =
    (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i+1,j+1)->Red) ||
    (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i,j+1)->Red) ;
    ImageOut(i*3,j*3+2)->Red = ImageIn(i-1,j)->Red ==
    ImageIn(i,j+1)->Red ?
    ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
    ImageOut(i*3+1,j*3+2)->Red =
    (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i+1,j+1)->Red) ||
    (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red &&
     ImageIn(i,j)->Red != ImageIn(i,j+1)->Red) ;
    ImageOut(i*3+2,j*3+2)->Red = ImageIn(i,j+1)->Red ==
    ImageIn(i-1,j)->Red ?
    ImageIn(i-1,j)->Red : ImageIn(i,j)->Red;
}
else
{
    ImageOut(3*i,3*j))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i+1,3*j))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i+2,3*j))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i,3*j+1))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i+1,3*j+1))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i+2,3*j+1))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i,3*j+2))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i+1,3*j+2))->Red = ImageIn(i,j)->Red;
    ImageOut(3*i+2,3*j+2))->Red = ImageIn(i,j)->Red;
}
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*3,j*3))->Green = ImageIn(i-1,j)->Green ==
        ImageIn(i,j-1)->Green ?
        ImageIn(i,j-1)->Green : ImageIn(i,j)->Green;
    ImageOut(i*3+1,j*3))->Green =
        (ImageIn(i,j-1)->Green == ImageIn(i,j+1)->Green &&
        ImageIn(i,j)->Green != ImageIn(i+1,j-1)->Green)
    || (ImageIn(i,j-1)->Green == ImageIn(i,j+1)->Green &&
        ImageIn(i,j)->Green != ImageIn(i-1,j-1)->Green)
    ? ImageIn(i,j-1)->Green : ImageIn(i,j)->Green;
    ImageOut(i*3+2,j*3))->Green = ImageIn(i,j-1)->Green ==
        ImageIn(i+1,j)->Green ?
        ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
    ImageOut(i*3,j*3+1))->Green =
        (ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green &&
        ImageIn(i,j)->Green != ImageIn(i-1,j+1)->Green)
    || (ImageIn(i,j+1)->Green == ImageIn(i,j+1)->Green &&
        ImageIn(i,j)->Green != ImageIn(i,j-1)->Green)
    ? ImageIn(i,j-1)->Green : ImageIn(i,j)->Green;
    ImageOut(i*3+1,j*3+1))->Green = ImageIn(i,j-1)->Green ==
        ImageIn(i+1,j)->Green ?
        ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;
    ImageOut(i*3+2,j*3+1))->Green =
        (ImageIn(i,j-1)->Green == ImageIn(i+1,j)->Green &&
        ImageIn(i,j)->Green != ImageIn(i+1,j+1)->Green)
    || (ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green &&
        ImageIn(i,j)->Green != ImageIn(i+1,j-1)->Green)
    ? ImageIn(i,j)->Green : ImageIn(i,j)->Green;
}
69
ImageOut(i*3,j*3+2)->Green = ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green ? ImageIn(i-1,j)->Green : ImageIn(i,j)->Green;

ImageOut(i*3+1,j*3+2)->Green = (ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green && ImageIn(i,j)->Green != ImageIn(i+1,j+1)->Green) || (ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green && ImageIn(i,j)->Green != ImageIn(i,j+1)->Green) ? ImageIn(i,j+1)->Green : ImageIn(i,j)->Green;

ImageOut(i*3+2,j*3+2)->Green = ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green ? ImageIn(i+1,j)->Green : ImageIn(i,j)->Green;

} else
{
  ImageOut(3*i,3*j)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i+1,3*j)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i+2,3*j)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i,3*j+1)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i+1,3*j+1)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i+2,3*j+1)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i+1,3*j+2)->Green = ImageIn(i,j)->Green;
  ImageOut(3*i+2,3*j+2)->Green = ImageIn(i,j)->Green;

  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*3,j+3)->Blue = ImageIn(i-1,j)->Blue == ImageIn(i,j-1)->Blue ? ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;
    ImageOut(i*3+1,j*3)->Blue = (ImageIn(i-1,j)->Blue == ImageIn(i,j-1)->Blue && ImageIn(i,j)->Blue != ImageIn(i+1,j-1)->Blue) || (ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue && ImageIn(i,j)->Blue != ImageIn(i-1,j+1)->Blue) ? ImageIn(i,j+1)->Blue : ImageIn(i,j)->Blue;
    ImageOut(i*3+2,j*3)->Blue = ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue ? ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;
  }
}
ImageOut(i+1,j)->Blue : ImageIn(i,j)->Blue;

ImageOut(i*3,j*3+1)->Blue =
(ImageIn(i-1,j)->Blue == ImageIn(i-1,j+1)->Blue && ImageIn(i,j)->Blue != ImageIn(i,j+1)->Blue) || (ImageIn(i+1,j)->Blue == ImageIn(i,j+1)->Blue && ImageIn(i,j)->Blue != ImageIn(i+1,j+1)->Blue)

? ImageIn(i-1,j)->Blue : ImageIn(i,j)->Blue;

ImageOut(i*3+1,j*3+1)->Blue = ImageIn(i,j)->Blue;

ImageOut(i*3+2,j*3+1)->Blue =
(ImageIn(i,j-1)->Blue == ImageIn(i+1,j)->Blue && ImageIn(i,j)->Blue != ImageIn(i+1,j+1)->Blue) || (ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue && ImageIn(i,j)->Blue != ImageIn(i+1,j-1)->Blue)

? ImageIn(i+1,j)->Blue : ImageIn(i,j)->Blue;

ImageOut(i*3,j*3+2)->Blue = ImageIn(i,j+1)->Blue ==
ImageIn(i,j+1)->Blue ? ImageIn(i,j)->Blue : ImageIn(i,j+1)->Blue;

ImageOut(i*3+1,j*3+2)->Blue =
(ImageIn(i-1,j)->Blue == ImageIn(i,j+1)->Blue && ImageIn(i,j)->Blue != ImageIn(i,j+1)->Blue) || (ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue && ImageIn(i,j)->Blue != ImageIn(i,j+1)->Blue)

? ImageIn(i,j+1)->Blue : ImageIn(i,j)->Blue;

ImageOut(i*3+2,j*3+2)->Blue = ImageIn(i,j+1)->Blue ==
ImageIn(i,j+1)->Blue ? ImageIn(i,j)->Blue : ImageIn(i,j+1)->Blue;

}

else
{

ImageOut(3*i,3*j)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i+1,3*j)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i+2,3*j)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i,3*j+1)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i+1,3*j+1)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i+2,3*j+1)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i,3*j+2)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i+1,3*j+2)->Blue = ImageIn(i,j)->Blue;
ImageOut(3*i+2,3*j+2)->Blue = ImageIn(i,j)->Blue;

}
```c
// Interpolation with RGB:
void rogers2xAlgorithm(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-1,j)->Blue != ImageIn(i+1,j)->Blue 
                && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue 
                && ImageIn(i-1,j)->Green != ImageIn(i,j+1)->Green 
                && ImageIn(i,j-1)->Green != ImageIn(i+1,j)->Green )
            {
                ImageOut(i*2,j*2)->Red =  
                    (ImageIn(i-1,j-1)->Red + 4*ImageIn(i-1,j)->Red +  
                    ImageIn(i-1,j+1)->Red + ImageIn(i+1,j)->Red)/8;  
                ImageOut(i*2+1,j*2)->Red =  
                    (ImageIn(i-1,j-1)->Red + ImageIn(i-1,j+1)->Red +  
                    ImageIn(i-1,j+1)->Red + 4*ImageIn(i+1,j)->Red)/8;  
                ImageOut(i*2,j*2+1)->Red =  
                    (4*ImageIn(i,j)-Red + ImageIn(i,j+1)->Red +  
                    ImageIn(i,j+1)->Red + ImageIn(i,j+1)->Red)/8;  
                ImageOut(i*2+1,j*2+1)->Red =  
                    (4*ImageIn(i,j)-Red + ImageIn(i,j+1)->Red +  
                    ImageIn(i,j+1)->Red + ImageIn(i,j+1)->Red)/8;  
            }
            else
            {
                ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;   
                ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;  
                ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;  
                ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;  
            }
        }
    }
    if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1 
        && j != ImageIn.TellHeight()-1 
        && ImageIn(i,j-1)->Blue != ImageIn(i,j+1)->Blue 
        && ImageIn(i-1,j)->Blue != ImageIn(i+1,j)->Blue 
        && ImageIn(i,j-1)->Red != ImageIn(i,j+1)->Red 
        && ImageIn(i-1,j)->Red != ImageIn(i+1,j)->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 )
{
    ImageOut(i*2,j*2)->Blue =
    (ImageIn(i-1,j-1)->Blue + ImageIn(i,j-1)->Blue +
    ImageIn(i-1,j)->Blue + 4*ImageIn(i,j)->Blue)/8;
    ImageOut(i*2+1,j*2)->Blue =
    (ImageIn(i,j-1)->Blue + ImageIn(i+1,j-1)->Blue +
    4*ImageIn(i,j)->Blue + ImageIn(i+1,j)->Blue)/8;
    ImageOut(i*2,j*2+1)->Blue =
    (ImageIn(i-1,j)->Blue + 4*ImageIn(i,j)->Blue +
    ImageIn(i-1,j+1)->Blue + ImageIn(i,j+1)->Blue)/8;
    ImageOut(i*2+1,j*2+1)->Blue =
    (4*ImageIn(i,j)->Blue + ImageIn(i+1,j)->Blue +
    ImageIn(i,j+1)->Blue + ImageIn(i+1,j+1)->Blue)/8;
}
else
{
    ImageOut(2*i,2*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(2*i+1,2*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
    ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;
}

if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
&& ImageIn(i,j-1)->Green != ImageIn(i,j+1)->Green
&& ImageIn(i-1,j)->Green != ImageIn(i+1,j)->Green
&& ImageIn(i-1,j)->Red != ImageIn(i+1,j)->Red
&& 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)->Green =
    (ImageIn(i-1,j-1)->Green + ImageIn(i,j-1)->Green +
    ImageIn(i-1,j)->Green + 4*ImageIn(i,j)->Green)/8;
    ImageOut(i*2+1,j*2)->Green =
    (ImageIn(i,j-1)->Green + ImageIn(i+1,j-1)->Green +
    4*ImageIn(i,j)->Green + ImageIn(i+1,j)->Green)/8;
    ImageOut(i*2,j*2+1)->Green =
    (ImageIn(i-1,j)->Green + 4*ImageIn(i,j)->Green +
    ImageIn(i-1,j+1)->Green + ImageIn(i,j+1)->Green)/8;
    ImageOut(i*2+1,j*2+1)->Green =
    (4*ImageIn(i,j)->Green + ImageIn(i+1,j)->Green +
    ImageIn(i,j+1)->Green + ImageIn(i+1,j+1)->Green)/8;
}
else
{
    ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
    ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
}
ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
}
}
}
}
}
}

// Interpolation with RGB:
void rogers3xAlgorithm(BMP ImageIn, BMP &ImageOut)
{
    int NewWidth = (int) ( ImageIn.TellWidth() * 3);
    int NewHeight = (int) ( ImageIn.TellHeight() * 3);
    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 )
            {
                if (ImageIn(i,j-1)->Red == ImageIn(i-1,j)->Red)
                    ImageOut(i*3,j*3)->Red = ImageIn(i,j-1)->Red;
                else
                    ImageOut(i*3,j*3)->Red = // E0
                        (ImageIn(i,j-1)->Red +
                        ImageIn(i-1,j)->Red + 2*ImageIn(i,j)->Red)/4;
            }
            else
            {
                if (ImageIn(i,j-1)->Red == ImageIn(i+1,j)->Red)
                    ImageOut(i*3+2,j*3)->Red = ImageIn(i,j-1)->Red;
                else
                    ImageOut(i*3+2,j*3)->Red = // E2
                        (ImageIn(i,j-1)->Red +
                        2*ImageIn(i,j)->Red + ImageIn(i+1,j)->Red)/4;
            }
            if (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red)
                ImageOut(i*3,j*3+2)->Red = ImageIn(i-1,j)->Red;
            else
            {
                ImageOut(i*3,j*3+2)->Red = // E6
                    (ImageIn(i-1,j)->Red +
                    2*ImageIn(i,j)->Red + ImageIn(i,j+1)->Red)/4;
            }
            if (ImageIn(i+1,j)->Red == ImageIn(i,j+1)->Red)
                ImageOut(i*3+2,j*3+2)->Red = ImageIn(i+1,j)->Red;
            else
            {
                ImageOut(i*3+2,j*3+2)->Red = // E8
                    (ImageIn(i+1,j)->Red +
                    2*ImageIn(i,j)->Red + ImageIn(i,j+1)->Red)/4;
            }
        }
    }
}
else {
    ImageOut(i*3+2, j*3+2)->Red = // E8
    (2*ImageIn(i, j)->Red + ImageIn(i+1, j)->Red +
     ImageIn(i, j+1)->Red)/4;
}
}

ImageOut(i*3+1, j*3+1)->Red = ImageIn(i, j)->Red; // E4

ImageOut(i*3+1, j*3)->Red = // E1
(2*ImageIn(i, j-1)->Red + 6*ImageIn(i, j)->Red)/8;
ImageOut(i*3, j*3+1)->Red = // E3
(2*ImageIn(i-1, j)->Red + 6*ImageIn(i, j)->Red)/8;
ImageOut(i*3+2, j*3+1)->Red = // E5
(2*ImageIn(i+1, j)->Red + 6*ImageIn(i, j)->Red)/8;
ImageOut(i*3+1, j*3+2)->Red = // E7
(2*ImageIn(i, j+1)->Red + 6*ImageIn(i, j)->Red)/8;
}
else {

    ImageOut(3*i, 3*j)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i+1, 3*j)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i+2, 3*j)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i, 3*j+1)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i+1, 3*j+1)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i+2, 3*j+1)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i+1, 3*j+2)->Red = ImageIn(i, j)->Red;
    ImageOut(3*i+2, 3*j+2)->Red = ImageIn(i, j)->Red;
}

if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
    && j != ImageIn.TellHeight()-1
    && ImageIn(i, j-1)->Blue != ImageIn(i, j+1)->Blue
    && ImageIn(i-1, j)->Blue != ImageIn(i+1, j)->Blue )
{
    if (ImageIn(i, j-1)->Blue == ImageIn(i-1, j)->Blue)
        ImageOut(i*3+3, j*3)->Blue = ImageIn(i, j-1)->Blue;
    else {
        ImageOut(i*3, j*3)->Blue = // E0
        (ImageIn(i, j-1)->Blue +
         ImageIn(i-1, j)->Blue + 2*ImageIn(i, j)->Blue)/4;
    }
    if (ImageIn(i, j-1)->Blue == ImageIn(i+1, j)->Blue)
        ImageOut(i*3+2, j*3)->Blue = ImageIn(i, j-1)->Blue;
    else {
        ImageOut(i*3+2, j*3)->Blue = // E2
        (ImageIn(i, j-1)->Blue +
         2*ImageIn(i, j)->Blue + ImageIn(i+1, j)->Blue)/4;
    }
    if (ImageIn(i-1, j)->Blue == ImageIn(i, j+1)->Blue)
else {
    ImageOut(i*3,j*3+2)->Blue = ImageIn(i-1,j)->Blue + 2*ImageIn(i,j)->Blue +
    ImageIn(i,j+1)->Blue)/4;
}

if (ImageIn(i+1,j)->Blue == ImageIn(i,j+1)->Blue)
    ImageOut(i*3+2,j*3+2)->Blue = ImageIn(i+1,j)->Blue;
else {
    ImageOut(i*3+2,j*3+2)->Blue = (ImageIn(i-1,j)->Blue +
    ImageIn(i,j+1)->Blue)/4;
}

ImageOut(i*3+1,j*3+1)->Blue = ImageIn(i,j)->Blue; // E5

ImageOut(i*3+1,j*3)->Blue = (2*ImageIn(i,j-1)->Blue + 6*ImageIn(i,j)->Blue)/8;
ImageOut(i*3,j*3+1)->Blue = (2*ImageIn(i-1,j)->Blue + 6*ImageIn(i,j)->Blue)/8;
ImageOut(i*3+2,j*3+1)->Blue = (2*ImageIn(i+1,j)->Blue + 6*ImageIn(i,j)->Blue)/8;

ImageOut(i*3+1,j*3+2)->Blue = (2*ImageIn(i,j+1)->Blue + 6*ImageIn(i,j)->Blue)/8;
}
else {
    ImageOut(3*i,3*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+1,3*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+2,3*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+1,3*j+1)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+1,3*j+1)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+2,3*j+1)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+1,3*j+2)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+1,3*j+2)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+2,3*j+2)->Blue = ImageIn(i,j)->Blue;
    ImageOut(3*i+2,3*j+2)->Blue = ImageIn(i,j)->Blue;
}

if (i != 0 && j != 0 && i != ImageIn.TellWidth()-1
    && j != ImageIn.TellHeight()-1
    && ImageIn(i-1,j-1)->Green != ImageIn(i,j+1)->Green
    && ImageIn(i-1,j-1)->Green != ImageIn(i+1,j)->Green )
    { if (ImageIn(i,j-1)->Green == ImageIn(i-1,j)->Green)
        ImageOut(i*3,j*3)->Green = ImageIn(i,j-1)->Green;
    else {
        ImageOut(i*3,j*3)->Green = (ImageIn(i,j-1)->Green +
        ImageIn(i,j)->Green + 2*ImageIn(i-1,j)->Green)/4;
    }
if (ImageIn(i, j-1)->Green == ImageIn(i+1, j)->Green)
    ImageOut(i*3+2, j*3)->Green = ImageIn(i, j-1)->Green;
else {
    ImageOut(i*3+2, j*3)->Green = \(\text{ImageIn}(i, j-1)\rightarrow\text{Green} + 2\times\text{ImageIn}(i, j)\rightarrow\text{Green} + \text{ImageIn}(i+1, j)\rightarrow\text{Green})/4;\)
}

if (ImageIn(i-1, j)->Green == ImageIn(i, j+1)->Green)
    ImageOut(i*3, j*3+2)->Green = ImageIn(i-1, j)->Green;
else {
    ImageOut(i*3, j*3+2)->Green = (\text{ImageIn}(i-1, j)\rightarrow\text{Green} + 2\times\text{ImageIn}(i, j)\rightarrow\text{Green} + \text{ImageIn}(i, j+1)\rightarrow\text{Green})/4;\)
}

if (ImageIn(i+1, j)->Green == ImageIn(i, j+1)->Green)
    ImageOut(i*3+2, j*3+2)->Green = ImageIn(i+1, j)->Green;
else {
    ImageOut(i*3+2, j*3+2)->Green = (2\times\text{ImageIn}(i, j)\rightarrow\text{Green} + 6\times\text{ImageIn}(i, j)\rightarrow\text{Green} + \text{ImageIn}(i, j+1)\rightarrow\text{Green})/8;\)
}

ImageOut(i*3+1, j*3+1)->Green = ImageIn(i, j)->Green; // E5

ImageOut(i*3+1, j*3)->Green = (2\times\text{ImageIn}(i, j-1)\rightarrow\text{Green} + 6\times\text{ImageIn}(i, j)\rightarrow\text{Green})/8;
ImageOut(i*3, j*3+1)->Green = \(\text{ImageIn}(i-1, j)\rightarrow\text{Green} + 6\times\text{ImageIn}(i, j)\rightarrow\text{Green})/8;\)
ImageOut(i*3+2, j*3+1)->Green = \(\text{ImageIn}(i+1, j)\rightarrow\text{Green} + 6\times\text{ImageIn}(i, j)\rightarrow\text{Green})/8;\)
ImageOut(i*3+1, j*3+2)->Green = \(2\times\text{ImageIn}(i, j+1)\rightarrow\text{Green} + 6\times\text{ImageIn}(i, j)\rightarrow\text{Green})/8;\)

else {
    ImageOut(3*i, 3*j)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+1, 3*j)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+2, 3*j)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+1, 3*j+1)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+1, 3*j+1)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+2, 3*j+1)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+1, 3*j+2)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+1, 3*j+2)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+2, 3*j+2)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
    ImageOut(3*i+2, 3*j+2)\rightarrow\text{Green} = ImageIn(i, j)\rightarrow\text{Green};
}

}
A.3 Rogers2x algorithm

```c
#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));


                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
```

78
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;
}

if (j % 10 == 0) cout << j << ", " << ImageOut.TellHeight() << endl;

void rogers2xAvgGFX_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 < ImageIn.TellHeight() ; j++)
    {
        for (int i=0 ; i < ImageIn.TellWidth() ; i++)
        {
            if (i != 0 && i != ImageIn.TellWidth()-1 &&
                j != 0 && j != ImageIn.TellHeight()-1)
            {

                //--------------//
                // p1 | p2 | p3 // p5 //--------------//
                // p4 | p5 | p6 // ---> // p01 | p02 //
                // p7 | p8 | p9 // // p03 | p04 //
                //--------------// //--------------//

                YUV p1, p2, p3, p4, p5, p6, p7, p8, p9, pol1, po2, pol3, po4;
                p1 = RGB2YUV(*ImageIn(i-1, j-1));
                p2 = RGB2YUV(*ImageIn(i, j-1));
                p3 = RGB2YUV(*ImageIn(i+1, j-1));
                p4 = RGB2YUV(*ImageIn(i-1, j));
                p5 = RGB2YUV(*ImageIn(i, j));
                p6 = RGB2YUV(*ImageIn(i+1, j));
                p7 = RGB2YUV(*ImageIn(i-1, j+1));
                p8 = RGB2YUV(*ImageIn(i, j+1));
                p9 = RGB2YUV(*ImageIn(i+1, j+1));

            }
        }
    }
}
\[ \text{po1.V} = \frac{(\text{p2.V} + \text{p4.V}) + 2\times \text{p5.V}}{4}; \]
\[ \text{po2.Y} = \frac{(\text{p2.Y} + \text{p6.Y}) + 2\times \text{p5.Y}}{4}; \]
\[ \text{po2.U} = \frac{(\text{p2.U} + \text{p6.U}) + 2\times \text{p5.U}}{4}; \]
\[ \text{po2.V} = \frac{(\text{p2.V} + \text{p6.V}) + 2\times \text{p5.V}}{4}; \]
\[ \text{po3.Y} = \frac{(\text{p4.Y} + \text{p8.Y}) + 2\times \text{p5.Y}}{4}; \]
\[ \text{po3.U} = \frac{(\text{p4.U} + \text{p8.U}) + 2\times \text{p5.U}}{4}; \]
\[ \text{po3.V} = \frac{(\text{p4.V} + \text{p8.V}) + 2\times \text{p5.V}}{4}; \]
\[ \text{po4.Y} = \frac{(\text{p6.Y} + \text{p8.Y}) + 2\times \text{p5.Y}}{4}; \]
\[ \text{po4.U} = \frac{(\text{p6.U} + \text{p8.U}) + 2\times \text{p5.U}}{4}; \]
\[ \text{po4.V} = \frac{(\text{p6.V} + \text{p8.V}) + 2\times \text{p5.V}}{4}; \]

\text{RGB RGBpo1} = \text{YUV2RGB(p01)};
\text{RGB RGBpo2} = \text{YUV2RGB(p02)};
\text{RGB RGBpo3} = \text{YUV2RGB(p03)};
\text{RGB RGBpo4} = \text{YUV2RGB(p04)};

\text{ImageOut}(i \times 2, j \times 2) -> Red = (\text{char}) \text{RGBpo1.R};
\text{ImageOut}(i \times 2, j \times 2) -> Green = (\text{char}) \text{RGBpo1.G};
\text{ImageOut}(i \times 2, j \times 2) -> Blue = (\text{char}) \text{RGBpo1.B};

\text{ImageOut}(i \times 2+1, j \times 2) -> Red = (\text{char}) \text{RGBpo2.R};
\text{ImageOut}(i \times 2+1, j \times 2) -> Green = (\text{char}) \text{RGBpo2.G};
\text{ImageOut}(i \times 2+1, j \times 2) -> Blue = (\text{char}) \text{RGBpo2.B};

\text{ImageOut}(i \times 2, j \times 2+1) -> Red = (\text{char}) \text{RGBpo3.R};
\text{ImageOut}(i \times 2, j \times 2+1) -> Green = (\text{char}) \text{RGBpo3.G};
\text{ImageOut}(i \times 2, j \times 2+1) -> Blue = (\text{char}) \text{RGBpo3.B};

\text{ImageOut}(i \times 2+1, j \times 2+1) -> Red = (\text{char}) \text{RGBpo4.R};
\text{ImageOut}(i \times 2+1, j \times 2+1) -> Green = (\text{char}) \text{RGBpo4.G};
\text{ImageOut}(i \times 2+1, j \times 2+1) -> Blue = (\text{char}) \text{RGBpo4.B};

\text{else}\{
\text{ImageOut}(2 \times i, 2 \times j) -> Red = \text{ImageIn}(i, j) -> Red;}
\text{ImageOut}(2 \times i+1, 2 \times j) -> Red = \text{ImageIn}(i, j) -> Red;}
\text{ImageOut}(2 \times i, 2 \times j+1) -> Red = \text{ImageIn}(i, j) -> Red;}
\text{ImageOut}(2 \times i+1, 2 \times j+1) -> Red = \text{ImageIn}(i, j) -> Red;}
\text{ImageOut}(2 \times i, 2 \times j) -> Blue = \text{ImageIn}(i, j) -> Blue;}
\text{ImageOut}(2 \times i+1, 2 \times j) -> Blue = \text{ImageIn}(i, j) -> Blue;}
\text{ImageOut}(2 \times i, 2 \times j+1) -> Blue = \text{ImageIn}(i, j) -> Blue;}
\text{ImageOut}(2 \times i+1, 2 \times j+1) -> Blue = \text{ImageIn}(i, j) -> Blue;}
\text{ImageOut}(2 \times i, 2 \times j) -> Green = \text{ImageIn}(i, j) -> Green;}
\text{ImageOut}(2 \times i+1, 2 \times j) -> Green = \text{ImageIn}(i, j) -> Green;}
\text{ImageOut}(2 \times i, 2 \times j+1) -> Green = \text{ImageIn}(i, j) -> Green;}
\text{ImageOut}(2 \times i+1, 2 \times j+1) -> Green = \text{ImageIn}(i, j) -> Green;\}
void rogers2xGFX_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 < ImageIn.TellHeight() ; j++)
    {
        for (int i=0 ; i < ImageIn.TellWidth() ; i++)
        {
            if (i != 0 && i != ImageIn.TellWidth()-1 &&
                j != 0 && j != ImageIn.TellHeight()-1)
            {

                YUV p1,p2,p3,p4,p5,p6,p7,p8,p9,po1,po2,po3,po4;
                p1 = RGB2YUV(*ImageIn(i-1, j-1));
                p2 = RGB2YUV(*ImageIn(i,j-1));
                p3 = RGB2YUV(*ImageIn(i+1,j-1));
                p4 = RGB2YUV(*ImageIn(i-1,j));
                p5 = RGB2YUV(*ImageIn(i,j));
                p6 = RGB2YUV(*ImageIn(i+1,j));
                p7 = RGB2YUV(*ImageIn(i-1,j+1));
                p8 = RGB2YUV(*ImageIn(i,j+1));
                p9 = RGB2YUV(*ImageIn(i+1,j+1));

                if (ImageIn(i-1,j)->Red == ImageIn(i,j-1)->Red
                    && ImageIn(i-1,j)->Blue == ImageIn(i,j-1)->Blue
                    && ImageIn(i-1,j)->Green == ImageIn(i,j-1)->Green)
                {
                }

                //--------------//
                // p1 | p2 | p3 // p5 //--------//
                // p4 | p5 | p6 // --> // po1 | po2 //
                // p7 | p8 | p9 // // po3 | po4 //
                //--------------//
                //--------//
            }
        }
    }

ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;

    if (j % 10 == 0) cout << j << " /" << ImageIn.TellHeight() << endl;
}
else
{
}

if (ImageIn(i,j-1)->Red == ImageIn(i+1,j)->Red
    && ImageIn(i,j-1)->Blue == ImageIn(i+1,j)->Blue
    && ImageIn(i,j-1)->Green == ImageIn(i+1,j)->Green)
{
}
else
{
}

if (ImageIn(i-1,j)->Red == ImageIn(i,j+1)->Red
    && ImageIn(i-1,j)->Blue == ImageIn(i,j+1)->Blue
    && ImageIn(i-1,j)->Green == ImageIn(i,j+1)->Green)
{
}
else
{
}

if (ImageIn(i,j+1)->Red == ImageIn(i+1,j)->Red
    && ImageIn(i,j+1)->Blue == ImageIn(i+1,j)->Blue
    && ImageIn(i,j+1)->Green == ImageIn(i+1,j)->Green)
{
}
else
{
}
RGB RGBpo1 = YUV2RGB(po1);
RGB RGBpo2 = YUV2RGB(po2);
RGB RGBpo3 = YUV2RGB(po3);
RGB RGBpo4 = YUV2RGB(po4);

ImageOut(i*2,j*2)->Red = (char) RGBpo1.R;
ImageOut(i*2,j*2)->Green = (char) RGBpo1.G;
ImageOut(i*2,j*2)->Blue = (char) RGBpo1.B;

ImageOut(i*2+1,j*2)->Red = (char) RGBpo2.R;
ImageOut(i*2+1,j*2)->Green = (char) RGBpo2.G;
ImageOut(i*2+1,j*2)->Blue = (char) RGBpo2.B;

ImageOut(i*2,j*2+1)->Red = (char) RGBpo3.R;
ImageOut(i*2,j*2+1)->Green = (char) RGBpo3.G;
ImageOut(i*2,j*2+1)->Blue = (char) RGBpo3.B;

ImageOut(i*2+1,j*2+1)->Red = (char) RGBpo4.R;
ImageOut(i*2+1,j*2+1)->Green = (char) RGBpo4.G;
ImageOut(i*2+1,j*2+1)->Blue = (char) RGBpo4.B;

} else {
    ImageOut(2*i,2*j)->Red = ImageIn(i,j)->Red;
    ImageOut(2*i+1,2*j)->Red = ImageIn(i,j)->Red;
    ImageOut(2*i,2*j+1)->Red = ImageIn(i,j)->Red;
    ImageOut(2*i+1,2*j+1)->Red = ImageIn(i,j)->Red;

    ImageOut(2*i,2*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(2*i+1,2*j)->Blue = ImageIn(i,j)->Blue;
    ImageOut(2*i,2*j+1)->Blue = ImageIn(i,j)->Blue;
    ImageOut(2*i+1,2*j+1)->Blue = ImageIn(i,j)->Blue;

    ImageOut(2*i,2*j)->Green = ImageIn(i,j)->Green;
    ImageOut(2*i+1,2*j)->Green = ImageIn(i,j)->Green;
    ImageOut(2*i,2*j+1)->Green = ImageIn(i,j)->Green;
    ImageOut(2*i+1,2*j+1)->Green = ImageIn(i,j)->Green;
}

if (j % 10 == 0) cout << j << "/" << ImageIn.TellHeight() << endl;

YUV YUV2YUV(RGBApixel In)
{
    YUV Out; // legge inn (int), raskere?
/* Factors for implementing in HW */
Out.U = (In.Blue - Out.Y)/2;  // U
Out.V = (In.Red - Out.Y)/2;  // V

/* ***Original transform factors***
Out.Lum = 0.299*ImageIn(x,y)->Red + 0.587*ImageIn(x,y)->Green +
  0.114*ImageIn(x,y)->Blue;  // Y
Out.Krom1 = (ImageIn(x,y)->Blue - Out.Lum)*0.565;  // U
Out.Krom2 = (ImageIn(x,y)->Red - Out.Lum)*0.713;  // V
*/
return Out;
}

RGB YUV2RGB(YUV In)
{
    RGB Out;

    /* Factors for implementing in HW */
    Out.R = 2*In.V + In.Y;
    Out.G = In.Y - In.U/2 - In.V/2;
    Out.B = In.Y + 2*In.U;

    /* ***Original transform factors***
    Out.R = 1.403*In.Krom2 + In.Lum;
    Out.G = In.Lum - 0.344*In.Krom1 - 0.714*In.Krom2;
    Out.B = In.Lum + 1.77*In.Krom1;
    */
    return Out;
}
# B Verilog code

## B.1 AXI Read module

```verilog
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;

/* 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;
```
output   ARVALID;
input    ARREADY;

/* Read data channel signals */
input [4:0]   RID;
input [63:0]  RDATA;
input [1:0]   RRESP;
input        RLAST;
output       RREADY;

/* Memory module signals */
input        stop_transfer;

/* RAM signals */
output       we;
output [63:0] di;

/* Input from APB */
input [31:0] start_addr;
input [15:0] mem_size_in;
input        start_scale;

/**************************************************************/
/* Register AXI inputs */
/**************************************************************/
reg          start_scale_r;
reg          reset_start_scale_r;
reg          first_addr_read;

/**************************************************************/
/* AXI Read register space */
/**************************************************************/
reg          araddr_r;
reg          arvalid_r;
reg          rready_r;
reg          we_r;

/**************************************************************/
/* Signals to control: */
/**************************************************************/
// Output: ARVALID, RREADY;
// [31:0] ARADDR;

/**************************************************************/
/* Signals to check: */
/**************************************************************/
// Input: RLAST, RVALID, ARREADY;
// [1:0] RRESP;
// [3:0] RID;
// [31:0] RDATA;

/**************************************************************/
always @(negedge ARESETn or posedge ACLK)
begin
  if (!ARESETn)
  begin
    rready_r <= 1'b0;
    we_r <= 1'b0;
    arvalid_r <= 1'b0;
    araddr_r <= 32'h00000000;
  //start_scale_r <= 1'b0;
    reset_start_scale_r <= 1'b0;
    first_addr_read <= 1'b1;
  end
  // elsif(posedge ACLK)
  else begin
    // Stop reading in new addresses?
    if (start_scale_r) // Start scale has been set.
      arvalid_r <= ~stop_transfer; // stop_transfer is a signal from mem_reg.
    // End of frame:
    if (araddr_r == start_addr + Burst_size*(Length+1)*mem_size_in ) begin
      arvalid_r <= 1'b0; // Next address not valid
      reset_start_scale_r <= 1'b1; // Ready for next frame
      first_addr_read <= 1'b1;
    end
  else begin
    // Increment address:
    if (ARREADY && arvalid_r) begin // equal sentence insted of counter
      araddr_r <= araddr_r + Burst_size*(Length+1); // araddr_r += 8*8
      if (araddr_r == start_addr + Burst_size*(Length+1)*((mem_size_in-1))
        arvalid_r <= 1'b0;
      end
      reset_start_scale_r <= 1'b0;
    end
  // Start reading, first valid address:
  if (start_scale_r && first_addr_read) // Insure 1 access
  begin
    araddr_r <= start_addr; // ADDR from APB
    arvalid_r <= 1'b1; // Valid address
    first_addr_read <= 1'b0; // Insure that reset/start happens in 1 clk cycle
  end
always @(posedge start_scale or posedge reset_start_scale_r) begin
    if (start_scale)
        start_scale_r <= 1;
    if (reset_start_scale_r)
        start_scale_r <= 0;
end

/*****************************************************************************
* Drive outputs
******************************************************************************/

// Constants:
assign ARBURST = Burst;
assign ARLOCK = Lock;
assign ARSIZE = Size;
assign ARPROT = Prot;
assign ARLEN = Length;
assign ARCACHE = Cache;
assign ARID = Id;
assign di = RDATA; // di (RAM) is always RDATA, use WR_EN signal.

// Registers:
assign ARADDR = araddr_r;
assign ARVALID = arvalid_r;
assign RREADY = 1’b1;
assign we = RVALID;
endmodule
B.2 AXI Write module

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, almost_empty, empty, rd_en,
                  data_count, start_scale, dest_addr, mem_size_out, irq_vs, frame_finished);

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

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;

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;
/* Write data channel signals */
output [4:0] WID;
output [63:0] WDATA;
output [7:0] WSTRB;
output WLAST;
output WVALID;
input WREADY;

/* Write response channel signals */
input [4:0] BID;
input [1:0] BRESP;
input BVALID;
output BREADY;

/* FIFO signals */
input empty;
input almost_empty;
input [63:0] dout;
input [9:0] data_count;
output rd_en;

/* APB signals */
input start_scale;
input [31:0] dest_addr;
input [15:0] mem_size_out;
output irq_vs; // Interrupt: Finished scaling

/* Input from scale module (Last data written to FIFO) */
input frame_finished;

/************************************************************
* AXI Read register space
************************************************************/
/* Register for input signals */
reg start_scale_r;
reg frame_finished_r;

/* Registers for output signals */
reg awvalid_r;
reg wlast_r;
reg wvalid_r;
reg rd_en_r;
reg [31:0] awaddr_r;
reg addr_finished_r;

/* Variables */
reg [2:0] countToLast; // count to Length (Length=8) 7 down to 0.
/* Process */
always @(negedge ARESETn or posedge ACLK)
beginn
if (!ARESETn)
begin
awvalid_r <= 1'b0;
avaddr_r <= 32'h00010000;
wlast_r <= 1'b0;
wvalid_r <= 1'b0;
rd_en_r <= 1'b0;
addr_finished_r <= 1'b0;
frame_finished_r <= 1'b0;
//start_scale_r <= 1'b0;
reset_start_scale_r <= 1'b0;

countToLast <= Length; // Length-1, count down to 0.
burst_in_FIFO <= 1'b0;
first_FIFO_read <= 1'b1;
first_addr_write <= 1'b1;
end
else // elsif(posedge ACLK)
begin
reset_start_scale_r <= 0;

// Increment address:
if (AWREADY && awvalid_r)
awaddr_r <= awaddr_r + Burst_size*(Length+1); // awaddr_r += 8*8
// Finished writing address (AWVALID <= 0):
if (awaddr_r == ( dest_addr + Burst_size*(Length+1)*mem_size_out ) ) begin
awvalid_r <= 1'b0;
addr_finished_r <= 1'b1;
end
else
awvalid_r <= 1'b0;
awaddr_r <= 32'h00010000;

// get rid of the first FIFO data...
first_FIFO_read; // get rid of the first FIFO data...
// Output: AWVALID, WLAST, WVALID, BREADY;
// [31:0] AWADDR, WDATA;
// Input: AWREADY, WREADY, BVALID;
// [1:0] BRESP;
// [3:0] BID;
// Signals to control:
// Signals to check:
// Read data from FIFO: (Moved up now...)
wlast_r <= 1'b0;  // Default 0: Not last byte read.
rd_en_r <= 1'b0;  //!! Default 0: Not enable read on FIFO.

if (countToLast == 0)
wlast_r <= 1;

// Read number of burst:
if (WREADY && wvalid_r) begin // wvalid_r
rd_en_r <= 1'b1;  // Tell FIFO that data is read
if (countToLast == 0) begin
wlast_r <= 0;
burst_in_FIFO <= 0;
wvalid_r <= 0;
countToLast <= Length;
rd_en_r <= 0;
end
else
if (countToLast == 1)
wlast_r <= 1'b1;
countToLast <= countToLast - 1;
end

// If FIFO holds at least a burst, write on AXI Write bus:
if (data_count > Burst_size-1) begin
burst_in_FIFO <= 1;
if (first_FIFO_read) begin // FIFO holds reset value first
wvalid_r <= 0;
first_FIFO_read <= 0;
end
else
wvalid_r <= 1;
if (WREADY) // If AXI master is ready, set read enable on FIFO
rd_en_r <= 1;
end

// Signal from scale module that last data packet is written to FIFO.
if (frame_finished) begin
frame_finished_r <= 1;
reset_start_scale_r <= 1;
end

// Start reading to destination, first valid address:
if (start_scale_r && first_addr_write) begin // Insure 1 access

awaddr_r <= dest_addr;
avvalid_r <= 1'b1;
frame_finished_r <= 1'b0;
addr_finished_r <= 1'b0;
countToLast <= Length;
//start_scale_r <= 1'b1;
burst_in_FIFO <= 1'b0;
//first_FIFO_read <= 1'b1;
first_addr_write <= 1'b0;
end
end
end
always @(posedge start_scale or posedge reset_start_scale_r)
begin
if (start_scale)
start_scale_r <= 1;
if (reset_start_scale_r)
start_scale_r <= 0;
end
*****************************************************************************/
Assign output constants:
assign AWBURST = Burst;
assign AWLOCK = Lock;
assign AWSIZE = Size;
assign AWPROT = Prot;
assign AWLEN = Length;
assign AWCACHE = Cache;
assign AWID = Id;
assign WID = Id;
assign WSTRB = Strb;
assign WDATA = dout; // WDATA is wired to output of FIFO.
assign BREADY = 1'b1; // No use of response channel
// Registers:
assign AWVALID = awvalid_r;
assign AWADDR = awaddr_r;
assign WLAST = wlast_r;
assign WVALID = wvalid_r;
assign rd_en = WREADY && burst_in_FIFO; // rd_en_r;
// IRQ when: FIFO is empty, finished writing addresses, scale module finished.
assign irq_vs = (almost_empty || empty) && addr_finished_r && frame_finished_r;
endmodule
B.3 APB module

```verilog
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
```

if (apb_en) begin
  paddr_r <= PADDR;
pwdata_r <= PWDATA;
end

always @(posedge PCLK or negedge PRESETn) begin
  if (!PRESETn)
    pwrite_r <= 1'b0;
  else // if(apb_en) pwrite_r <= PSEL && PWRITE && !PENABLE else the same
    pwrite_r <= ( apb_en ? PSEL && PWRITE && !PENABLE : pwrite_r );
end

*****************************************************************************/
* APB register space and interrupt handling.
******************************************************************************/
'define APB_START_ADDR 12'h00 // Physical start address
'define APB_DEST_ADDR 12'h04 // Physical destination address
'define APB_MEM_IN_SIZE 12'h08 // Set size to read
'define APB_MEM_OUT_SIZE 12'h0c // Set size to write
'define APB_START_SCALE 12'h10 // To start scaling
'define APB_FRAME_W_IN 12'h14 // Set input frame width
'define APB_FRAME_W_OUT 12'h18 // Set output frame width
'define APB_FRAME_H_OUT 12'h1c // Set output frame height
'define APB_TOTAL_COUNT 12'h20 // Read total count

reg [31:0] start_addr_r;
reg [31:0] dest_addr_r;
reg [15:0] mem_size_in_r;
reg [15:0] mem_size_out_r;
reg start_scale_r;
reg [10:0] line_width_in_r;
reg [10:0] frame_width_out_r;
reg [10:0] frame_height_out_r;
reg [31:0] read_data;
reg [23:0] total_clk_count;
reg en_count;
reg irq_vs_r;

wire [11:0] cfg_addr = paddr_r[11:0];

// APB read
always @(paddr_r or start_addr_r or dest_addr_r or mem_size_in_r or
  mem_size_out_r or start_scale_r or line_width_in_r or frame_width_out_r or
  frame_height_out_r or total_clk_count or cfg_addr) begin
  case (cfg_addr)
    'APB_START_ADDR:   read_data = start_addr_r;
    'APB_DEST_ADDR:    read_data = dest_addr_r;
    'APB_MEM_IN_SIZE:  read_data = { 16'b0, mem_size_in_r };
    'APB_MEM_OUT_SIZE: read_data = { 16'b0, mem_size_out_r };
  endcase
end
always @(posedge PCLK or negedge PRESETn) begin
  if (!PRESETn) begin
    start_addr_r <= 32'hfffffff;
    dest_addr_r <= 32'hfffffff;
    mem_size_in_r <= 16'h0000;
    mem_size_out_r <= 16'h0000;
    start_scale_r <= 1'b0;
    line_width_in_r <= 11'h000;
    frame_width_out_r <= 11'h000;
    frame_height_out_r <= 11'h000;
    total_clk_count <= 24'h000000;
    //en_count <= 1'b0;
  end
  else begin
    if (pwrite_r && cfg_addr == 'APB_START_ADDR)
      start_addr_r <= pwdata_r;
    if (pwrite_r && cfg_addr == 'APB_DEST_ADDR)
      dest_addr_r <= pwdata_r;
    if (pwrite_r && cfg_addr == 'APB_START_SCALE)
      start_scale_r <= pwdata_r[0];
    if (pwrite_r && cfg_addr == 'APB_MEM_IN_SIZE)
      mem_size_in_r <= pwdata_r[15:0];
    if (pwrite_r && cfg_addr == 'APB_MEM_OUT_SIZE)
      mem_size_out_r <= pwdata_r[15:0];
    if (pwrite_r && cfg_addr == 'APB_FRAME_W_IN)
      line_width_in_r <= pwdata_r[10:0];
    if (pwrite_r && cfg_addr == 'APB_FRAME_W_OUT)
      frame_width_out_r <= pwdata_r[10:0];
    if (pwrite_r && cfg_addr == 'APB_FRAME_H_OUT)
      frame_height_out_r <= pwdata_r[10:0];
    irq_vs_r <= irq_vs;
  end
end
// Count total clock periods used to scale:
if (en_count)
  total_clk_count <= total_clk_count + 1;
end
end

// Total clk count process:
always @(posedge start_scale_r or posedge irq_vs_r)
begin
  if (start_scale_r)
    en_count <= 1'b1;
  if (irq_vs_r)
    en_count <= 1'b0;
end

****************************************************************************
* Drive outputs
******************************************************************************/
assign PRDATA = read_data;
assign PREADY = 1'b1;
assign PSLVERR = 1'b0;
assign start_addr = start_addr_r;
assign dest_addr = dest_addr_r;
assign mem_size_in = mem_size_in_r;
assign mem_size_out = mem_size_out_r;
assign start_scale = start_scale_r;
assign line_width_in = line_width_in_r;
assign frame_width_out = frame_width_out_r;
assign frame_height_out = frame_height_out_r;
endmodule
B.4 Memory Control module (mem_reg)

```vhdl
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

// 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; // Update registers
input FIFO_full;
output reg_ready; // Tell if registers have data ready
output [255:0] line1;
output [255:0] line2;

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

---

99
* Register space

*******************************************************************************/
reg stop_transfer_r;
reg [ADDR_WIDTH:0] ar_r; // one extra bit to insure that aw_r > ar_r
reg [ADDR_WIDTH:0] aw_r; // one extra bit to insure that aw_r > ar_r
reg reg_ready_r;

wire [7:0] LINE_WIDTH;

// Counters:
reg [7:0] count_line_width; // max: 1920/8 = 240 (h'F0)
reg [2:0] count_first_four;
reg [1:0] reg_pos;
reg [3:0] line_number;
reg [15:0] dummy_mem_count;

// Variable:
reg on_line1;

// Wire l1 and l2 to line1 and line2 registers:
reg [63:0] l1 [0:3];
reg [63:0] l2 [0:3];

/*******************************************************************************/
/* Assign constant */
assign LINE_WIDTH = line_width_in [10:3]; // Set from APB. line width / 8.

/* Process */
always @(negedge ARESETn or posedge ACLK)
begin
if (!ARESETn)
begin
  // Output registers:
  stop_transfer_r <= 1'b0;
  aw_r <= 0;
  ar_r <= 0;
  reg_ready_r <= 1'b0;
  //line1_r <= 0;
  //line2_r <= 0;
  // Counters:
  count_line_width <= 0;
  count_first_four <= 0;
  reg_pos <= 0;
  line_number <= 0;
  dummy_mem_count <= 0;
  // Variable:
  on_line1 <= 1;
end
end
// elsif(posedge ACLK)
else
begin

  // Update address on RAM when a write_enable occurs:
  if (write_enable) begin // same as we on RAM
    aw_r <= aw_r + 1;
  end

  // Initial start: (first line?) // Latency of 2 when updating ar (address read)
  if (count_first_four < 4) begin // != 4?
    reg_ready_r <= 0;
    if (aw_r != ar_r && aw_r-1 != ar_r) begin
      l1[count_first_four] <= ram_data;
      ar_r <= ar_r + 1;
      count_first_four <= count_first_four + 1; // Reset?
    reg_ready_r <= 1;
    count_line_width <= 4;
    on_line1 <= 1;
    end
  end

  // Normal sequence:
  else if (aw_r != ar_r && reg_shift && !FIFO_full) begin // change reg_shift in scale125!
    // Read part of Linel from RAM
    if (on_line1) begin // reg_ready_r
      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;
    dummy_mem_count <= dummy_mem_count + 1; // For testing..

    if (count_line_width == LINE_WIDTH-1) begin
      count_line_width <= 0;
      line_number <= line_number + 1;
    end
if (line_number == 0 || line_number == 2 ||
line_number == 4 || line_number == 6 ||
line_number == 8 || line_number == 10 ||
line_number == 12 || line_number == 13) begin
    ar_r <= ar_r - LINE_WIDTH - LINE_WIDTH + 1;
end

else if (line_number == 14) begin
    ar_r <= ar_r + 1;
    line_number <= 0;
else
    ar_r <= ar_r - LINE_WIDTH + 1;
end

else
    ar_r <= ar_r - LINE_WIDTH + 1; // Read first line next

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

else if (aw_r == ar_r || FIFO_full)
    reg_ready_r <= 0;
else // reg_shift = 0, calculate new without changing registers.
    reg_ready_r <= 1;
end

// Controll of RAM (ar must be behind aw):
if (on_line1) begin
    if (aw_r > ar_r + 256) // random: find minimum! (256 addr ahead....)
        stop_transfer_r <= 1; // Stop reading in new addresses no AXI read
    else
        stop_transfer_r <= 0;
end
if (ar_r[ADDR_WIDTH-1:0] == 0) begin
    aw_r[ADDR_WIDTH] <= 0;
    ar_r[ADDR_WIDTH] <= 0;
end
end // elsif(posedge ACLK)
end // always

*****************************************************************************
* Drive outputs
***************************************************************************/
assign stop_transfer = stop_transfer_r;
assign aw = aw_r[ADDR_WIDTH-1:0];
assign ar = ar_r[ADDR_WIDTH-1:0];
assign reg_ready = reg_ready_r;
assign line1 = {l1[0], l1[1], l1[2], l1[3]};
assign line2 = {l2[0], l2[1], l2[2], l2[3]};
endmodule
B.5 Scale 1.25 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);

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 have data ready
output reg_shift; // Update registers

// APB signals */
input [10:0] frame_width_o;
input [10:0] frame_height_o;
input [255:0] line1_r, line2_r;
reg wr_en_r;
reg frame_finished_r;
reg reg_shift_r;
reg [63:0] fifo_data;
wire [7:0] FRAME_WIDTH_OUT;
wire [10:0] FRAME_HEIGHT_OUT;

// Counters:
reg [2:0] line_number; // 5 line number used. Toggle between 5 lines.
reg [2:0] line_count; // 5 different ways to calculate output per line.
reg [8:0] horizontal_count; // +1 for every 5 output pixels. Max: 256
reg [7:0] vertical_count; // +1 for every 5 output lines in frame. (144)
reg [1:0] yuv_count;
reg [15:0] dummy_count;
reg [15:0] dummy_FIFO_full_count;
reg last_data_packet; // Ensure that last data is read.

// Input register for task (avg_func):
reg [4:0] s1 [0:7];
reg [4:0] s2 [0:7];
reg [4:0] s3 [0:7];
reg [4:0] s4 [0:7];
reg [7:0] pix1 [0:7];
reg [7:0] pix2 [0:7];
reg [7:0] pix3 [0:7];
reg [7:0] pix4 [0:7];

// Wire p1 and p2 to line1 and line2 register:
wire [7:0] p1 [0:31];
wire [7:0] p2 [0:31];

assign {p1[7], p1[6], p1[5], p1[4], p1[3], p1[2], p1[1], p1[0], p1[15], p1[14],
p1[13], p1[12], p1[11], p1[10], p1[9], p1[8], p1[23], p1[22], p1[21], p1[20],
p1[19], p1[18], p1[17], p1[16], p1[31], p1[30], p1[29], p1[28], p1[27], p1[26],
p1[25], p1[24]} = line1_r;

assign {p2[7], p2[6], p2[5], p2[4], p2[3], p2[2], p2[1], p2[0], p2[15], p2[14],
p2[13], p2[12], p2[11], p2[10], p2[9], p2[8], p2[23], p2[22], p2[21], p2[20],
p2[19], p2[18], p2[17], p2[16], p2[31], p2[30], p2[29], p2[28], p2[27], p2[26],
p2[25], p2[24]} = line2_r;
/* Assign constants */
assign FRAME_WIDTH_OUT = frame_width_o[10:3];
assign FRAME_HEIGHT_OUT = frame_height_o;

/** Function / Task declaration
******************************************************************************/
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;
beg
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]; // divide by 16
end
end
endtask

/* Process */
  avg_func(s1[0],s2[0],s3[0],s4[0], pix1[0],pix2[0],pix3[0],pix4[0],
           fifo_data[7:0]);
  avg_func(s1[1],s2[1],s3[1],s4[1], pix1[1],pix2[1],pix3[1],pix4[1],
           fifo_data[15:8]);
  avg_func(s1[2],s2[2],s3[2],s4[2], pix1[2],pix2[2],pix3[2],pix4[2],
           fifo_data[23:16]);
  avg_func(s1[3],s2[3],s3[3],s4[3], pix1[3],pix2[3],pix3[3],pix4[3],
           fifo_data[31:24]);
  avg_func(s1[4],s2[4],s3[4],s4[4], pix1[4],pix2[4],pix3[4],pix4[4],
           fifo_data[39:32]);
  avg_func(s1[5],s2[5],s3[5],s4[5], pix1[5],pix2[5],pix3[5],pix4[5],
           fifo_data[47:40]);
end
fifo_data[47:40]);
146 avg_func(s1[6],s2[6],s3[6],s4[6], pix1[6],pix2[6],pix3[6],pix4[6],
147 fifo_data[55:48]);
148 end
149
150 /* Process */
151 always @(negedge ARESETn or posedge ACLK)
152 begin
153 if (!ARESETn)
154 begin
155 // Output registers:
156 reg_shift_r <= 1'b0;
157 frame_finished_r <= 1'b0;
158 wr_en_r <= 1'b0;
159 // Counters:
160 horizontal_count <= 0;
161 vertical_count <= 0;
162 line_number <= 3'b000;
163 line_count <= 3'b000;
164 last_data_packet <= 0;
165 dummy_count <= 0;
166 dummy_FIFO_full_count <= 0;
167 yuv_count <= 0;
168 // Reset intern registers? (+ 123 LUTS (20 % more), + 14MHz (4,3%))
169 //s1 <= 0; s2 <= 0; s3 <= 0; s4 <= 0;
170 //pix1 <= 0; pix2 <= 0; pix3 <= 0; pix4 <= 0;
171 end
172
173 // else if(posedge ACLK)
174 else begin
175 frame_finished_r <= 1'b0; // Default value
176 if (FIFO_full) // If FIFO is full hold last reg_shift value
177 reg_shift_r <= reg_shift_r;
178 else
179 reg_shift_r <= 1'b1; // Default value
180 wr_en_r <= 1'b0; // Default value
181
182 // Start
183 if (reg_ready || last_data_packet)
184 begin
185 wr_en_r <= 1'b1;
186 last_data_packet <= 0;
187
188 end
if (line_count == 3'b000) begin
  pix1[0] <= p1[0]; pix2[0] <= p1[1]; pix3[0] <= p2[0]; pix4[0] <= p2[1];

  line_count <= line_count + 1;
  reg_shift_r <= 1'b0;
end

else if (line_count == 3'b001) begin
  pix1[0] <= p1[6]; pix2[0] <= p1[7]; pix3[0] <= p2[6]; pix4[0] <= p2[7];

  line_count <= line_count + 1;

  // SHIFT IN NEW DATA IN [255:192]
  reg_shift_r <= 1'b1;
end

else if (line_count == 3'b010) begin
  pix1[0] <= p1[12]; pix2[0] <= p1[13]; pix3[0] <= p2[12]; pix4[0] <= p2[13];

line_count <= line_count + 1;

// SHIFT IN NEW DATA IN {191:128}
reg_shift_r <= 'b1;
end

else if (line_count == 3'b011) begin
    pix1[0] <= p1[18]; pix2[0] <= p1[19]; pix3[0] <= p2[18]; pix4[0] <= p2[19];

line_count <= line_count + 1;

// SHIFT IN NEW DATA IN {127:64}
reg_shift_r <= 'b1;
end

else begin // line_count == 4
    pix1[0] <= p1[25]; pix2[0] <= p1[26]; pix3[0] <= p2[25]; pix4[0] <= p2[26];
end
269 p2[29];
274
275 line_count <= 3'b000;
276
277 // SHIFT IN NEW DATA IN [63:0]
278 reg_shift_r <= 1'b1;
279
end

//----------------------------LINE 0-----------------------------------//

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[5] <= a00; s2[5] <= a00; s3[5] <= a00; s4[5] <= a00;
    s1[6] <= a00; s2[6] <= a00; s3[6] <= a00; s4[6] <= a00;
    s1[7] <= a00; s2[7] <= a00; s3[7] <= a00; s4[7] <= a00;
  end
else if (line_count == 3'b001) begin
  s1[0] <= a12; s2[0] <= a04; s3[0] <= a00; s4[0] <= a00;
  s1[1] <= a00; s2[1] <= a16; s3[1] <= a00; s4[1] <= a00;
  s1[2] <= a16; s2[2] <= a00; s3[2] <= a00; s4[2] <= a00;
  s1[7] <= a16; s2[7] <= a00; s3[7] <= a00; s4[7] <= a00;
end
else if (line_count == 3'b010) begin
  s1[0] <= a04; s2[0] <= a12; s3[0] <= a00; s4[0] <= a00;
  s1[1] <= a08; s2[1] <= a08; s3[1] <= a00; s4[1] <= a00;

end
```verilog
//----------------------------LINE 1-----------------------------------//
else if (line_number == 3'b001) begin
  if (line_count == 3'b000) begin
    s1[0] <= a04; s2[0] <= a00; s3[0] <= a12; s4[0] <= a0;  
    s1[1] <= a01; s2[1] <= a03; s3[1] <= a03; s4[1] <= a9;  
  end
end

else if (line_count == 3'b011) begin
  s1[0] <= a00; s2[0] <= a16; s3[0] <= a00; s4[0] <= a0;  
  s1[1] <= a16; s2[1] <= a00; s3[1] <= a0; s4[1] <= a0;  
end

else begin  // line_count == 4
  s1[0] <= a08; s2[0] <= a08; s3[0] <= a00; s4[0] <= a0;  
  s1[1] <= a12; s2[1] <= a04; s3[1] <= a00; s4[1] <= a0;  
  s1[7] <= a00; s2[7] <= a16; s3[7] <= a00; s4[7] <= a0;  
end

end

//----------------------------LINE 1-----------------------------------//
else if (line_number == 3'b001) begin
  if (line_count == 3'b000) begin
    s1[0] <= a04; s2[0] <= a00; s3[0] <= a12; s4[0] <= a0;  
    s1[1] <= a01; s2[1] <= a03; s3[1] <= a03; s4[1] <= a9;  
  end
end
```
else if (line_count == 3'b001) begin

    s1[0] <= a03; s2[0] <= a01; s3[0] <= a09; s4[0] <= a03;
    s1[1] <= a00; s2[1] <= a04; s3[1] <= a00; s4[1] <= a12;
    s1[7] <= a04; s2[7] <= a00; s3[7] <= a12; s4[7] <= a00;
end

else if (line_count == 3'b010) begin

    s1[0] <= a01; s2[0] <= a03; s3[0] <= a03; s4[0] <= a09;
    s1[1] <= a02; s2[1] <= a02; s3[1] <= a06; s4[1] <= a06;
end

else if (line_count == 3'b011) begin

    s1[0] <= a00; s2[0] <= a04; s3[0] <= a00; s4[0] <= a12;
end

else begin // line_count == 4

    s1[0] <= a02; s2[0] <= a02; s3[0] <= a06; s4[0] <= a06;
    s1[1] <= a03; s2[1] <= a01; s3[1] <= a09; s4[1] <= a03;
end
```cpp
//----------------------------LINE 2-----------------------------------//
else if (line_number == 3'b010) begin
  if (line_count == 3'b000) begin
    s1[0] <= a08; s2[0] <= a00; s3[0] <= a08; s4[0] <= a00;
    s1[1] <= a02; s2[1] <= a06; s3[1] <= a02; s4[1] <= a06;
  end
  else if (line_count == 3'b001) begin
    s1[0] <= a06; s2[0] <= a02; s3[0] <= a06; s4[0] <= a02;
    s1[1] <= a00; s2[1] <= a08; s3[1] <= a00; s4[1] <= a08;
  end
  else if (line_count == 3'b010) begin
    s1[0] <= a02; s2[0] <= a06; s3[0] <= a02; s4[0] <= a06;
    s1[3] <= a00; s2[3] <= a08; s3[3] <= a00; s4[3] <= a08;
  end
  else if (line_count == 3'b011) begin
    s1[0] <= a00; s2[0] <= a08; s3[0] <= a00; s4[0] <= a08;
    s1[1] <= a08; s2[1] <= a08; s3[1] <= a08; s4[1] <= a08;
    s1[5] <= a00; s2[5] <= a00; s3[5] <= a00; s4[5] <= a00;
  end
```
end

else begin // line_count == 4

s1[0] <= a04; s2[0] <= a04; s3[0] <= a04; s4[0] <= a04;
s1[1] <= a06; s2[1] <= a02; s3[1] <= a06; s4[1] <= a02;
s1[2] <= a00; s2[2] <= a08; s3[2] <= a00; s4[2] <= a08;
s1[3] <= a08; s2[3] <= a00; s3[3] <= a08; s4[3] <= a00;
s1[7] <= a00; s2[7] <= a08; s3[7] <= a00; s4[7] <= a08;
end
end

//----------------------------LINE 3-----------------------------------//
else if (line_number == 3'b011) begin
  if (line_count == 3'b000) begin
    s1[0] <= a12; s2[0] <= a00; s3[0] <= a04; s4[0] <= a00;
s1[1] <= a03; s2[1] <= a09; s3[1] <= a01; s4[1] <= a03;
  end
else if (line_count == 3'b001) begin
  s1[0] <= a09; s2[0] <= a03; s3[0] <= a03; s4[0] <= a01;
s1[1] <= a00; s2[1] <= a12; s3[1] <= a00; s4[1] <= a04;
s1[7] <= a12; s2[7] <= a00; s3[7] <= a04; s4[7] <= a00;
end
else if (line_count == 3'b010) begin
  s1[0] <= a03; s2[0] <= a09; s3[0] <= a01; s4[0] <= a03;
s1[1] <= a06; s2[1] <= a06; s3[1] <= a02; s4[1] <= a02;
end

else if (line_count == 3'b011) begin

  s1[0] <= a00; s2[0] <= a12; s3[0] <= a00; s4[0] <= a04;
s1[1] <= a00; s2[1] <= a00; s3[1] <= a04; s4[1] <= a00;
s1[3] <= a06; s2[3] <= a06; s3[3] <= a02; s4[3] <= a00;

end

else begin // line_count == 4

  s1[0] <= a06; s2[0] <= a06; s3[0] <= a02; s4[0] <= a02;
s1[1] <= a00; s2[1] <= a00; s3[1] <= a16; s4[1] <= a00;

end

else begin // line_number == 4

  if (line_count == 3'b000) begin

    s1[0] <= a00; s2[0] <= a00; s3[0] <= a16; s4[0] <= a00;
s1[1] <= a00; s2[1] <= a00; s3[1] <= a04; s4[1] <= a12;
s1[2] <= a00; s2[2] <= a00; s3[2] <= a08; s4[2] <= a08;
s1[5] <= a00; s2[5] <= a00; s3[5] <= a16; s4[5] <= a00;
s1[7] <= a00; s2[7] <= a00; s3[7] <= a08; s4[7] <= a08;

  end

end

//----------------------------LINE 4-----------------------------------//
else if (line_count == 3'b001) begin

    s1[0] <= a00; s2[0] <= a00; s3[0] <= a12; s4[0] <= a04;
    s1[1] <= a00; s2[1] <= a00; s3[1] <= a00; s4[1] <= a16;
    s1[2] <= a00; s2[2] <= a00; s3[2] <= a16; s4[2] <= a00;
    s1[7] <= a00; s2[7] <= a00; s3[7] <= a16; s4[7] <= a00;

end

else if (line_count == 3'b010) begin

    s1[0] <= a00; s2[0] <= a00; s3[0] <= a04; s4[0] <= a12;
    s1[1] <= a00; s2[1] <= a00; s3[1] <= a08; s4[1] <= a08;
    s1[3] <= a00; s2[3] <= a00; s3[3] <= a00; s4[3] <= a16;

end

else if (line_count == 3'b011) begin

    s1[0] <= a00; s2[0] <= a00; s3[0] <= a00; s4[0] <= a16;
    s1[1] <= a00; s2[1] <= a00; s3[1] <= a12; s4[1] <= a04;
    s1[2] <= a00; s2[2] <= a00; s3[2] <= a00; s4[2] <= a16;
    s1[3] <= a00; s2[3] <= a00; s3[3] <= a08; s4[3] <= a08;
    s1[5] <= a00; s2[5] <= a00; s3[5] <= a00; s4[5] <= a16;

end

else begin // line_count == 4

    s1[0] <= a00; s2[0] <= a00; s3[0] <= a08; s4[0] <= a08;
    s1[1] <= a00; s2[1] <= a00; s3[1] <= a12; s4[1] <= a04;
    s1[2] <= a00; s2[2] <= a00; s3[2] <= a00; s4[2] <= a16;
    s1[3] <= a00; s2[3] <= a00; s3[3] <= a16; s4[3] <= a00;
    s1[7] <= a00; s2[7] <= a00; s3[7] <= a00; s4[7] <= a16;

end
end

// ------------------------ Check horizontal ------------------------//
if (horizontal_count == FRAME_WIDTH_OUT-1) begin // A line is finished.
    horizontal_count <= 0;
    horizontal_count <= 0;
end
else
    line_number <= line_number + 1;

// ------------------------ Check Vertical ------------------------//
if (vertical_count == FRAME_HEIGHT_OUT-1) begin
    vertical_count <= 0;
    frame_finished_r <= 1'b1;
    yuv_count <= 2;
    yuv_count <= 0;
else
    yuv_count <= yuv_count + 1;
end
else
    vertical_count <= vertical_count + 1;
end // if (FIFO is not full AND registers are ready)
if (wr_en_r)
    dummy_count <= dummy_count + 1; // used for simulation
if ((horizontal_count == FRAME_WIDTH_OUT-1) &&
    (vertical_count == FRAME_HEIGHT_OUT-1)) begin
    last_data_packet <= 1;
end
end // elsif(posedge ACLK)
end // always

always @(posedge FIFO_full) begin
    dummy_FIFO_full_count <= dummy_FIFO_full_count + 1; // used for simulation
end
******************************************************************************
* Drive outputs
******************************************************************************
assign din   = fifo_data;
assign wr_en  = wr_en_r;
assign frame_finished = last_data_packet; //frame_finished_r
assign reg_shift = reg_shift_r;
endmodule
B.6 Scale 1.875 module

module scale1875(ACLK, ARESETn, din, FIFO_full, wr_en, frame_finished, reg_ready, reg_shift, line1_r, line2_r, frame_width_o, frame_height_o);

// 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 have data ready
output reg_shift; // Update registers

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

// Register space
input [255:0] line1_r, line2_r;
reg wr_en_r;
reg frame_finished_r;
reg reg_shift_r;
reg [63:0] fifo_data;
wire [7:0] FRAME_WIDTH_OUT;
wire [10:0] FRAME_HEIGHT_OUT;

// Counters:
reg [3:0] line_number; // 5 line number used. Toggle between 5 lines.
reg [3:0] line_count; // 5 different ways to calculate output per line.
reg [8:0] horizontal_count; // +1 for every 5 output pixels. Max: 256
reg [10:0] vertical_count; // +1 for every 5 output lines in frame. (144)
reg [1:0] yuv_count;
reg [3:0] x_count;
reg [3:0] y_count;
reg [3:0] y_reg;
reg [15:0] dummy_count;
reg last_data_packet; // Ensure that last data is read.

// Input register for task (odd_func):
reg [3:0] x_r [0:7]; // changed from 4:0...
reg [7:0] pix1 [0:7];
reg [7:0] pix2 [0:7];
reg [7:0] pix3 [0:7];
reg [7:0] pix4 [0:7];

// Wire p1 and p2 to line1 and line2 register:
wire [7:0] p1 [0:31];
wire [7:0] p2 [0:31];

assign {p1[7], p1[6], p1[5], p1[4], p1[3], p1[2], p1[1], p1[0], p1[15], p1[14],
p1[13], p1[12], p1[11], p1[10], p1[9], p1[8], p1[23], p1[22], p1[21], p1[20],
p1[19], p1[18], p1[17], p1[16], p1[31], p1[30], p1[29], p1[28], p1[27], p1[26],
p1[25], p1[24]} = line1_r;
assign {p2[7], p2[6], p2[5], p2[4], p2[3], p2[2], p2[1], p2[0], p2[15], p2[14],
p2[13], p2[12], p2[11], p2[10], p2[9], p2[8], p2[23], p2[22], p2[21], p2[20],
p2[19], p2[18], p2[17], p2[16], p2[31], p2[30], p2[29], p2[28], p2[27], p2[26],
p2[25], p2[24]} = line2_r;

/* Assign constants */
assign FRAME_WIDTH_OUT = frame_width_o [10:3];
assign FRAME_HEIGHT_OUT = frame_height_o;

/****************************************************************************
* Function / Task declaration
******************************************************************************/
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;
// 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];
end
else begin // (x_r and y_r are odd)
temp_res = temp1 + temp2 + temp3 + temp4;
po = temp_res[13:6];
end
endtask

/* Process */
begin
odd_func(x_r[0], y_reg, pix1[0], pix2[0], pix3[0], pix4[0], fifo_data [7:0]);
odd_func(x_r[1], y_reg, pix1[1], pix2[1], pix3[1], pix4[1], fifo_data [15:8]);
odd_func(x_r[2], y_reg, pix1[2], pix2[2], pix3[2], pix4[2], fifo_data [23:16]);
odd_func(x_r[3], y_reg, pix1[3], pix2[3], pix3[3], pix4[3], fifo_data [31:24]);
odd_func(x_r[4], y_reg, pix1[4], pix2[4], pix3[4], pix4[4], fifo_data [39:32]);
odd_func(x_r[5], y_reg, pix1[5], pix2[5], pix3[5], pix4[5], fifo_data [47:40]);
odd_func(x_r[6], y_reg, pix1[6], pix2[6], pix3[6], pix4[6], fifo_data [55:48]);
odd_func(x_r[7], y_reg, pix1[7], pix2[7], pix3[7], pix4[7], fifo_data [63:56]);
end
/* Process */
always @(negedge ARESETn or posedge ACLK)
begin
if (!ARESETn)
begin
    // Output registers:
    reg_shift_r <= 1'b0;
    frame_finished_r <= 1'b0;
    wr_en_r <= 1'b0;
    // Counters:
    horizontal_count <= 0;
    vertical_count <= 0;
    line_number <= 3'b000;
    line_count <= 3'b000;
    last_data_packet <= 0;
    dummy_count <= 0;
    yuv_count <= 0;
    y_count <= 0;
    y_reg <= 0;
    // Reset intern registers? (+ 123 LUTS (20 % more), + 14MHz (4,3%))
    //s1 <= 0; s2 <= 0; s3 <= 0; s4 <= 0;
    //pix1 <= 0; pix2 <= 0; pix3 <= 0; pix4 <= 0;
end
else
begin
    frame_finished_r <= 1'b0; // Default value
    if (FIFO_full)
        reg_shift_r <= reg_shift_r; // Hold value when FIFO is full
    else
        reg_shift_r <= 1'b1; // Default value
    wr_en_r <= 1'b0; // Default value
end
// Start
if (reg_ready || last_data_packet)
begin

wr_en_r <= 1'b1;
last_data_packet <= 0;

line_count <= line_count + 1;
y_reg <= y_count; // hold previous value of y_count in y_reg

case(line_count)
  4'b0000: begin
    pix1[0] <= p1[0]; pix2[0] <= p1[1];
    pix1[1] <= p1[0]; pix2[1] <= p1[1];

    pix3[0] <= p2[0]; pix4[0] <= p2[1];
    pix3[1] <= p2[0]; pix4[1] <= p2[1];
    x_r[0] = 0; x_r[1] = 1; x_r[2] = 2; x_r[3] = 3;

    reg_shift_r <= 1'b0;
  end

  4'b0001: begin
    pix1[0] <= p1[4]; pix2[0] <= p1[5];

    pix3[0] <= p2[4]; pix4[0] <= p2[5];
end

x_r[0] = 8; x_r[1] = 9; x_r[2] = 10; x_r[3] = 11;

// SHIFT IN NEW DATA IN {255:192}
reg_shift_r <= 1'b1;
end

4'b0010: begin
pix1[0] <= p1[8]; pix2[0] <= p1[9];
pix3[0] <= p2[8]; pix4[0] <= p2[9];
x_r[0] = 1; x_r[1] = 2; x_r[2] = 3; x_r[3] = 4;
reg_shift_r <= 1'b0;
end

4'b0011: begin
pix1[0] <= p1[12]; pix2[0] <= p1[13];
pix3[0] <= p2[12]; pix4[0] <= p2[13];

x_r[0] = 9; x_r[1] = 10; x_r[2] = 11; x_r[3] = 12;

// SHIFT IN NEW DATA IN [191:128]
reg_shift_r <= 1'b1;
end

4'b0100: begin
pix1[0] <= p1[17]; pix2[0] <= p1[18];
pix3[0] <= p2[17]; pix4[0] <= p2[18];
x_r[0] = 2; x_r[1] = 3; x_r[2] = 4; x_r[3] = 5;
reg_shift_r <= 1'b0;
end

4'b0101: begin
pix1[0] <= p1[21]; pix2[0] <= p1[22];
pix3[0] <= p2[21]; pix4[0] <= p2[22];
 x_r[0] = 10; x_r[1] = 11; x_r[2] = 12; x_r[3] = 13;

 // SHIFT IN NEW DATA IN {127:64}
 reg_shift_r <= 1'b1;
 end

 4'b0110: begin
 pix1[0] <= p1[25]; pix2[0] <= p1[26];
 pix3[0] <= p2[25]; pix4[0] <= p2[26];
 x_r[0] = 3; x_r[1] = 4; x_r[2] = 5; x_r[3] = 6;
 reg_shift_r <= 1'b0;
 end

 4'b0111: begin // 7
 pix1[0] <= p1[29]; pix2[0] <= p1[30];
 pix3[0] <= p2[29]; pix4[0] <= p2[30];

x_r[0] = 11; x_r[1] = 12; x_r[2] = 13; x_r[3] = 14;
x_r[4] = 0; x_r[5] = 1; x_r[6] = 2; x_r[7] = 3;

// SHIFT IN NEW DATA IN [63:0]
reg_shift_r <= 1'b1;
end

4'b1000: begin
pix1[0] <= p1[2]; pix2[0] <= p1[3];

pix3[0] <= p2[2]; pix4[0] <= p2[3];

x_r[0] = 4; x_r[1] = 5; x_r[2] = 6; x_r[3] = 7;

reg_shift_r <= 1'b0;
end

4'b1001: begin
pix1[0] <= p1[6]; pix2[0] <= p1[7];

pix3[0] <= p2[6]; pix4[0] <= p2[7];

x_r[0] = 12; x_r[1] = 13; x_r[2] = 14; x_r[3] = 0;

// SHIFT IN NEW DATA IN {255:192}
reg_shift_r <= 1'b1;
end

4'b1010: begin
pix1[0] <= p1[10]; pix2[0] <= p1[11];

pix3[0] <= p2[10]; pix4[0] <= p2[11];

x_r[0] = 5; x_r[1] = 6; x_r[2] = 7; x_r[3] = 8;
reg_shift_r <= 1'b0;
end

4'b1011: begin
pix1[0] <= p1[14]; pix2[0] <= p1[15];
pix3[0] <= p2[14]; pix4[0] <= p2[15];
566
567 x_r[0] = 13; x_r[1] = 14; x_r[2] = 0; x_r[3] = 1;
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];
583 pix3[0] <= p2[19]; pix4[0] <= p2[20];
591
592 x_r[0] = 6; x_r[1] = 7; x_r[2] = 8; x_r[3] = 9;
594
595 reg_shift_r <= 1'b0;
596 end
597
598 4'b1101: begin
599 pix1[0] <= p1[23]; pix2[0] <= p1[24];
607 pix3[0] <= p2[23]; pix4[0] <= p2[24];  // 105

x_r[0] = 14; x_r[1] = 0; x_r[2] = 1; x_r[3] = 2;

// SHIFT IN NEW DATA IN [127:64]
reg_shift_r <= 1'b1;
end

4'b1110: begin // 14
    pix1[0] <= p1[27]; pix2[0] <= p1[28];
pix1[7] <= p1[31]; pix2[7] <= p1[0];

    pix3[0] <= p2[27]; pix4[0] <= p2[28];
pix3[7] <= p2[31]; pix4[7] <= p2[0]; // 120

    x_r[0] = 7; x_r[1] = 8; x_r[2] = 9; x_r[3] = 10;

    // SHIFT IN NEW DATA IN [63:0]
    reg_shift_r <= 1'b1;
    line_count <= 0;
end

endcase

//------------------------ Check horizontal -------------------------://
if (horizontal_count == FRAME_WIDTH_OUT-1) begin // A line is finished.
    horizontal_count <= 0;
    if (y_count == 14)
        y_count <= 0;
else

end

//----------------------------------

131
y_count <= y_count + 1;

if (line_number == 14) begin
    line_number <= 0;
end
else
    line_number <= line_number + 1;

//------------------------ Check Vertical --------------------------/
if (vertical_count == FRAME_HEIGHT_OUT-1) begin
    vertical_count <= 0;
    frame_finished_r <= 1'b1;
    if (yuv_count == 2)
        yuv_count <= 0;
    else
        yuv_count <= yuv_count + 1;
end
else
    vertical_count <= vertical_count + 1;
end
else
    horizontal_count <= horizontal_count + 1;

end // if (FIFO is not full AND registers are ready)

if (wr_en_r)
    dummy_count <= dummy_count + 1;

if (((horizontal_count == FRAME_WIDTH_OUT-1) ||
    (horizontal_count == FRAME_WIDTH_OUT-2) ||
    (horizontal_count == FRAME_WIDTH_OUT-3) ||
    (horizontal_count == FRAME_WIDTH_OUT-4) ) &&
    (vertical_count == FRAME_HEIGHT_OUT-1) ) begin
    if (yuv_count == 2) begin
        last_data_packet <= 1;
    end
end
end // elsif(posedge ACLK)
end // always

//***************************************************************
// Drive outputs
//***************************************************************
assign din = fifo_data;
assign wr_en = wr_en_r;
assign frame_finished = last_data_packet; //frame_finished_r
assign reg_shift = reg_shift_r;
endmodule
B.7 RAM module

```verilog
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
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.ucd 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

Phase 12.8 Global Placement
Phase 12.8  Global Placement (Checksum:fd741de6) REAL time: 17 mins 11 secs
Phase 13.29 Local Placement Optimization
Phase 13.29 Local Placement Optimization (Checksum:fd741de6) REAL time: 17 mins 11 secs
Phase 14.5  Local Placement Optimization
Phase 14.5  Local Placement Optimization (Checksum:fd741de6) REAL time: 17 mins 16 secs
Phase 15.18 Placement Optimization
Phase 15.18 Placement Optimization (Checksum:c0afacfe) REAL time: 26 mins 45 secs
Phase 16.5  Local Placement Optimization
Phase 16.5  Local Placement Optimization (Checksum:c0afacfe) REAL time: 26 mins 50 secs
Phase 17.34 Placement Validation
Phase 17.34 Placement Validation (Checksum:c0afacfe) REAL time: 26 mins 53 secs
Total REAL time to Placer completion: 27 mins
Total CPU time to Placer completion: 26 mins 58 secs
Running post-placement packing...
Writing output files...

Design Summary
Number of errors: 0
Number of warnings: 203

Slice Logic Utilization:
Number of Slice Registers: 23,321 out of 207,360 11%
   Number used as Flip Flops: 23,321
Number of Slice LUTs: 30,974 out of 207,360 14%
   Number used as logic: 26,740 out of 207,360 12%
   Number using O6 output only: 26,018
   Number using O5 output only: 210
   Number using O5 and O6: 512
Number used as Memory: 4,214 out of 54,720 7%
   Number used as Dual Port RAM: 4,211
   Number using O5 output only: 5
   Number using O5 and O6: 4,206
   Number used as Shift Register: 3
   Number using O6 output only: 3
Number used as exclusive route-thru: 20
Number of route-thrus: 257
Number using O6 output only: 228
Number using O5 output only: 29

Slice Logic Distribution:
Number of occupied Slices: 14,093 out of 51,840 27%
Number of LUT Flip Flop pairs used: 42,758
Number with an unused Flip Flop: 19,437 out of 42,758 45%
Number with an unused LUT: 11,784 out of 42,758 27%
Number of fully used LUT-FF pairs: 11,537 out of 42,758 26%
Number of unique control sets: 754
Number of slice register sites lost to control set restrictions: 923 out of 207,360 1%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element. The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails. OVERMAPPING of BRAM resources should be ignored if the design is over-mapped for a non-BRAM resource or if placement fails.
IO Utilization:
Number of bonded IOBs: 620 out of 1,200 51%
Number of LOCed IOBs: 620 out of 620 100%
IOB Flip Flops: 601

Specific Feature Utilization:
Number of BlockRAM/FIFO: 20 out of 288 6%
Number using BlockRAM only: 16
Number using FIFO only: 4
Number of 36k BlockRAM used: 13
Number of 18k BlockRAM used: 3
Number of 36k FIFO used: 4
Total Memory used (KB): 666 out of 10,368 6%
Number of BUFG/BUFGCTRLs: 8 out of 32 25%
Number used as BUFGs: 8
Number of DCM ADVs: 2 out of 12 16%
Number of DSP48Es: 32 out of 192 16%
Average Fanout of Non-Clock Nets: 4.07
Peak Memory Usage: 2059 MB
Total REAL time to MAP completion: 28 mins 10 secs
Total CPU time to MAP completion: 28 mins 8 secs
Mapping completed.
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 : -l
11 Mapper Version : virtex5 — $Revision: 1.51.18.1$
12 Mapped Date : Mon Jun 7 12:05:35 2010
13
14....

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

140
Phase 14.5 Local Placement Optimization
Phase 14.5 Local Placement Optimization (Checksum:22b69109) REAL time: 17 mins 16 secs
Phase 15.18 Placement Optimization
Phase 15.18 Placement Optimization (Checksum: cce083f6) REAL time: 27 mins 28 secs
Phase 16.5 Local Placement Optimization
Phase 16.5 Local Placement Optimization (Checksum: cce083f6) REAL time: 27 mins 33 secs
Phase 17.34 Placement Validation
Phase 17.34 Placement Validation (Checksum: cce083f6) REAL time: 27 mins 36 secs
Total REAL time to Placer completion: 27 mins 43 secs
Total CPU time to Placer completion: 27 mins 40 secs
Running post-placement packing...
Writing output files...

Design Summary:
Number of errors: 0
Number of warnings: 199
Slice Logic Utilization:
Number of Slice Registers: 23,372 out of 207,360 11%
Number used as Flip Flops: 23,372
Number of Slice LUTs: 32,640 out of 207,360 15%
Number used as logic: 28,389 out of 207,360 13%
Number using O6 output only: 27,411
Number using O5 output only: 233
Number using O5 and O6: 745
Number used as Memory: 4,214 out of 54,720 7%
Number used as Dual Port RAM: 4,211
Number using O5 output only: 4,206
Number using O6 output only: 5
Number used as Shift Register: 3
Number using O6 output only: 3
Number used as exclusive route-thru: 37
Number of route-thrus: 362
Number using O6 output only: 269
Number using O5 output only: 93

Slice Logic Distribution:
Number of occupied Slices: 14,407 out of 51,840 27%
Number of LUT Flip Flop pairs used: 43,962
Number with an unused Flip Flop: 20,590 out of 43,962 46%
Number with an unused LUT: 11,322 out of 43,962 25%
Number of fully used LUT-FF pairs: 12,050 out of 43,962 27%
Number of unique control sets: 755
Number of slice register sites lost to control set restrictions: 932 out of 207,360 1%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element.
The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.
OVERMAPPING of BRAM resources should be ignored if the design is over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
Number of bonded IOBs: 620 out of 1,200 51%
Number of LOCed IOBs: 620 out of 620 100%
IOB Flip Flops: 601
Specific Feature Utilization:

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of BlockRAM/FIFO</td>
<td>20 out of 288 (6%)</td>
<td></td>
</tr>
<tr>
<td>Number using BlockRAM only</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Number using FIFO only</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total primitives used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 36k BlockRAM used</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Number of 18k BlockRAM used</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Number of 36k FIFO used</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total Memory used (KB)</td>
<td>666 out of 10,368 (6%)</td>
<td></td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs</td>
<td>8 out of 32 (25%)</td>
<td></td>
</tr>
<tr>
<td>Number used as BUFGs</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Number of DCMADVs</td>
<td>2 out of 12 (16%)</td>
<td></td>
</tr>
<tr>
<td>Number of DSP48Es</td>
<td>48 out of 192 (25%)</td>
<td></td>
</tr>
</tbody>
</table>

Average Fanout of Non-Clock Nets: 4.04

Peak Memory Usage: 2137 MB

Total REAL time to MAP completion: 28 mins 56 secs

Total CPU time to MAP completion: 28 mins 52 secs

Mapping completed.

See MAP report file "fpga_map.mrp" for details.
### D.3 Scale 1.25 module - Synplify log file

#### Performance Summary

<table>
<thead>
<tr>
<th>Starting Clock Type</th>
<th>Requested Clock Frequency</th>
<th>Estimated Clock Frequency</th>
<th>Requested Clock Period</th>
<th>Estimated Clock Period</th>
<th>Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale125</td>
<td>ACLK inferred</td>
<td>1.0 MHz</td>
<td>347.3 MHz</td>
<td>1000.000</td>
<td>2.879</td>
</tr>
</tbody>
</table>

#### Clock Relationships

<table>
<thead>
<tr>
<th>Starting clock</th>
<th>Ending constraint slack</th>
<th>constraint slack</th>
<th>constraint slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale125</td>
<td>ACLK</td>
<td>scale125</td>
<td>ACLK</td>
</tr>
</tbody>
</table>

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

...
Resource Usage Report for scale125

Mapping to part: xc5vlx330ff1760-2

Cell usage:
- DSP48E 32 uses
- FDC 8 uses
- FDC 8 uses
- FDE 266 uses
- GND 1 use
- MUXCY 4 uses
- MU XCY _L 39 uses
- VOC 1 use
- XORCY 3 uses
- LUT1 30 uses
- LUT2 16 uses
- LUT3 3 uses
- LUT4 2 uses
- LUT5 3 uses
- LUT6 382 uses
- LUT6 _L 200 uses

I/O ports: 605
I/O primitives: 602
- IBUF 534 uses
- IBUF 1 use
- OB UF 67 uses
- BUF G 1 use

I/O Register bits: 0
Register bits not including I/Os: 294 (0%)

DSP48s: 32 of 192 (16%)

Global Clock Buffers: 1 of 32 (3%)

Number of unique control sets: 4
- C(ACL K _c ) , CLR (ARESETn_c _i ) , PRE(GND) , CE(N180_i ) : 8
- C(ACL K _c ) , CLR (ARESETn_c _i ) , PRE(GND) , CE(VOC) : 8
- C(ACL K _c ) , CLR (ARESETn_c _i ) , PRE(GND) , CE(we n_r6 ) : 12
- C(ACL K _c ) , CLR (GND) , PRE(GND) , CE(pix 1_0..0_sqmuxa ) : 266

Total load per clock:
- scale125|ACLK: 326

Mapping Summary:
- Total LUTs: 636 (0%)

0%: MF234 |Hierarchical island-based critical path report is located in
/home/rogska01/master/synplify/scale_125/rev_2/scale125.tah

Mapper successful!
Process took 0h:00m:25s realtime, 0h:00m:25s cputime
# Mon Jun 14 12:50:48 2010
D.4 Scale 1.875 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-1sf4.trondheim.arm.com
#Implementation: rev_1
#Mon Jun 14 13:00:02 2010
....

Performance Summary

Worst slack in design: -0.388

<table>
<thead>
<tr>
<th>Starting Clock Group</th>
<th>Requested Clock Frequency</th>
<th>Estimated Clock Frequency</th>
<th>Requested Clock Period</th>
<th>Estimated Clock Period</th>
<th>Slack</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale 1875</td>
<td>ACLK</td>
<td>396.8 MHz</td>
<td>343.9 MHz</td>
<td>2.520</td>
<td>2.908</td>
<td>-0.388</td>
</tr>
</tbody>
</table>

Clock Relationships

<table>
<thead>
<tr>
<th>Clocks</th>
<th>start</th>
<th>end</th>
<th>constraint slack</th>
<th>constraint slack</th>
<th>constraint slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale 1875</td>
<td>ACLK</td>
<td>scale 1875</td>
<td>ACLK</td>
<td>2.520</td>
<td>-0.388</td>
</tr>
</tbody>
</table>

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 ######
Resource Usage Report for scale1875

Mapping to part: xc5vlx330ff1760-1

Cell usage:

- DSP48E: 64 uses
- FDC: 9 uses
- FDCE: 166 uses
- FDE: 480 uses
- GND: 1 use
- MUXCY: 10 uses
- VXCCY: 79 uses
- VCC: 1 use
- XORCY: 68 uses
- LUT1: 48 uses
- LUT2: 30 uses
- LUT3: 9 uses
- LUT4: 172 uses
- LUT5: 303 uses
- LUT6: 909 uses
- LUT6.2: 53 uses

I/O ports: 605
I/O primitives: 602
IBUF: 534 uses
IBUF: 1 use
OBUF: 67 uses
BUFG: 1 use

I/O Register bits: 0
Register bits not including I/Os: 655 (0%)

DSP48s: 64 of 192 (33%)

Global Clock Buffers: 1 of 32 (3%)

Number of unique control sets:
- 4
- C(ACLK), CLR(ARESETn,c,i), PRE(GND), CE(wr_en_r6_i) : 155
- C(ACLK), CLR(GND), PRE(GND), CE(line_count_fast,fast,fast,RN18CV3[2]) : 480
- C(ACLK), CLR(ARESETn,c,i), PRE(GND), CE(N.25_i.i) : 11
- C(ACLK), CLR(ARESETn,c,i), PRE(GND), CE(VCC) : 9

Total load per clock:
- scale1875|ACLK: 679

Mapping Summary:
Total LUTs: 1524 (0%)

Mapper successful!

Process took 0h:01m:43s realtime, 0h:01m:42s cputime
# Mon Jun 14 13:01:52 2010

Hierarchical island-based critical path report is located in
/home/rogska01/master/synplify/scale_1875/rev_1/scale1875.tah

107
### D.5 Mem_reg 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-1sf4.trondheim.arm.com
- **#Implementation**: rev_1
- **#Date**: Mon Jun 14 13:07:20 2010

#### Performance Summary

- **Worst slack in design**: -0.752

<table>
<thead>
<tr>
<th>Starting Clock</th>
<th>Requested Clock Frequency</th>
<th>Estimated Clock Frequency</th>
<th>Requested Clock Period</th>
<th>Estimated Clock Period</th>
<th>Slack</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_reg</td>
<td>ACLK</td>
<td></td>
<td>234.8 MHz</td>
<td>199.6 MHz</td>
<td>4.259</td>
<td>5.011</td>
</tr>
</tbody>
</table>

#### Clock Relationships

<table>
<thead>
<tr>
<th>Starting Clock</th>
<th>Ending Clock</th>
<th>constraint slack</th>
<th>constraint slack</th>
<th>constraint slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_reg</td>
<td>ACLK</td>
<td>4.259</td>
<td>-0.752</td>
<td>No paths</td>
</tr>
</tbody>
</table>

**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

- **# # # # # END OF TIMING REPORT #####
Resource Usage Report for mem_reg

Mapping to part: xc5vlx330ff1760-1

Cell usage:

<table>
<thead>
<tr>
<th>Cell</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDC</td>
<td>54</td>
</tr>
<tr>
<td>FDCE</td>
<td>520</td>
</tr>
<tr>
<td>FDP</td>
<td>3</td>
</tr>
<tr>
<td>GND</td>
<td>1</td>
</tr>
<tr>
<td>MUXCY</td>
<td>9</td>
</tr>
<tr>
<td>MUXCY_L</td>
<td>46</td>
</tr>
<tr>
<td>VCC</td>
<td>1</td>
</tr>
<tr>
<td>XORCY</td>
<td>37</td>
</tr>
<tr>
<td>LUT1</td>
<td>28</td>
</tr>
<tr>
<td>LUT2</td>
<td>8</td>
</tr>
<tr>
<td>LUT3</td>
<td>17</td>
</tr>
<tr>
<td>LUT4</td>
<td>26</td>
</tr>
<tr>
<td>LUT5</td>
<td>29</td>
</tr>
<tr>
<td>LUT6</td>
<td>57</td>
</tr>
<tr>
<td>LUT6_2</td>
<td>7</td>
</tr>
</tbody>
</table>

I/O ports: 614
I/O primitives: 611
IBUF    | 76 uses
IBUFG   | 1 use
OBUF    | 534 uses
BUF     | 1 use

I/O Register bits: 0
Register bits not including I/Os: 577 (0%)

Number of unique control sets: 11

<table>
<thead>
<tr>
<th>Control Set</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(VCC)</td>
<td>54</td>
</tr>
<tr>
<td>C(ACLK) , CLR(GND) , PRE(ARESETn,c,i) , CE(VCC)</td>
<td>3</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(N_17)</td>
<td>8</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(12_0..1_sqmuxa)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(12_1..1_sqmuxa)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(12_2..1_sqmuxa)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(reg_pos1_sqmuxa)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(N_4)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(N_6)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(N_8)</td>
<td>64</td>
</tr>
<tr>
<td>C(ACLK) , CLR(ARESETn,c,i) , PRE(GND) , CE(N_10)</td>
<td>64</td>
</tr>
</tbody>
</table>

Total load per clock:

mem_reg|ACLK: 577

Mapping Summary:
Total LUTs: 172 (0%)

@N: MF234 | Hierarchical island–based critical path report is located in /home/rosken/mem_reg1875/rev_1/mem_reg1875.tah

Mapper successful!
Process took 0h:00m:28s realtime, 0h:00m:28s cputime
# Mon Jun 14 13:07:52 2010

#***************************************************************************