NTNU – Trondheim
Norwegian University of
Science and Technology

3D Perspective Video Scaling Effects on FPGA

Eivind Karlsen

Master of Science in Electronics
Submission date: June 2013
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Ove Brynestad, Cisco

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Abstract

The goal of this thesis was to design a video scaler able to do a perspective transform on a video stream. The scaler should be designed in VHDL and for FPGA, and the implementation should focus on achieving a low area while still doing a visually pleasing transformation. Additionally, the circuit should be able to operate in real time on high resolution video.

The thesis continues the work carried out in a pre-project on the same topic. There, several algorithms were introduced and implemented in Matlab. The thesis is also inspired by a state of the art polyphase scaler able to scale between rectangular video, which is implemented in VHDL.

Before the hardware unit was designed, software models of the relevant algorithms was created in Matlab. These models were used to compare the algorithms, and later to verify the hardware implementation. The comparison shows that some of the algorithms give high quality output, but are complex to design, and others give a lower visual quality for a much simpler implementation. This gives much flexibility to adapt the system to the resources available.

To focus on the core functionality, the simplest set of algorithms were chosen for the hardware implementation. This was implemented in VHDL and tested and synthesised. During testing, two bugs were found, one in the calculation of perspective factors, used to tune the transform, and one affecting the last two columns off the output frame. The first error only affects the initialization of the module from software. The second affects visual quality, and needs further investigation. Apart from these errors, the design fulfills the requirements. Additionally, the synthesis revealed that the design takes up very few logic elements.

Sammendrag

Målet med denne masteroppgaven var å designe en videoskalerer som er i stand til å utføre perspektivtransformeringer på en videostrøm. Skalereren skal designes for FPGA i VHDL, med et fokus på lavt arealforbruk og samtidig tilstrekkelig god visuell transformering. Designet må også kunne operere i sanntid på høy-oppløselige videostrømmer.

Oppgaven fortsetter arbeidet som ble utført i et forprosjekt med samme tema. Der ble flere algoritmer utforsket og implementert i Matlab. Oppgaven bygger også på en polyfaseskalerer som kan skalere rektangulære videostrømmer, som er implementert i VHDL.

Før implementering i VHDL, ble relevante algoritmer modellert i Matlab. Disse ble brukt til å sammenligne de forskjellige algoritmene, samt å verifisere VHDL-implementeringa. Sammenligninga viste at Det var stort sprik i algoritmene. Noen viste eksemplarisk kvalitet på skaleringen, men er komplekse å implementere, mens andre enklere algoritmer ofrer kvalitet mot lave implementeringskostnader. Dette gir gode muligheter til å tilpasse algoritmer etter tilgjengelige ressurser.

For å holde fokuset på kjernefunksjonaliteten til skalereren, ble de enkleste algoritmene valgt for VHDL implementering. To feil ble funnet ved designet: En i initialiseringa av skalereren fra software, og en feil ved det genererte bildet. Denne andre feilen påvirker skalererens visuelle kvalitet, og bør dermed utforskes videre. Bortsett fra dise feilene oppfølger designet kravene. Syntetisering av modulen viser i tillegg at den bruker veldig små ressurser på en FPGA.

Contents

Abstract **i**

Sammendrag **ii**

1 Introduction **1**
1.1 Problem Description 1
1.2 Previous work and Contributions 1
1.3 Structure 1
1.4 Relationship with Pre-project 2

2 Geometrical image operations on FPGA **3**
2.1 Forward Mapping 3
2.2 Reverse Mapping 3
2.3 Buffer Technology 3

3 Perspective Scaling Algorithms **5**
3.1 Interpolation 5
3.2 Mapping Algorithms 7

4 The Polyphase Scaler **11**
4.1 Algorithm 11
4.2 System Architecture 12
4.3 Relevancy for Perspective Scaler 14

5 Perspective Scaler Model **15**
5.1 Modular Architecture 15
5.2 Iterative Mapping Model 16
5.3 Quality Comparison 19

6 Hardware Implementation **23**
6.1 Framework 23
6.2 Core Perspective Scaler Design 23
6.3 Architecture Documentation 25
6.4 Potential for improvement 26

7 Performance Analysis and Verification **28**
7.1 Verification Plan 28
7.2 Test Results 29
7.3 Synthesis 29
7.4 Specifications and Summary 32

8 Conclusion **34**
8.1 Future Work 34

A Appendix **36**

References **37**

1 Introduction

A perspective scaler is a module that is able to transform an image or a steam of images (video) in three dimensions. This operation is generally carried out by graphics processing units to map textures onto three dimensional objects. The perspective scaling operation can also be used to do key-stoning or other three dimensional effects, like picture-in-picture effects.

Perspective image scaling can be seen as more generalized rectangular (or classical) image scaling. A rectangular scaler can change the size of an image in both dimensions, changing the magnification of the image. In a general image scaler, the output is some kind of function of the input image which does not need to be rectangular or linear.

A video scaler is an image scaler able to scale several images each second. A video scaler must make a new calculation, possibly including several multiplications, for every pixel in the video. Because of this complexity, and the necessity for real time solutions for certain applications, it is a task that benefits from a hardware implementation.

1.1 Problem Description

The following problem description is the basis for the thesis:

"A perspective video scaler should be created in VHDL and implemented on FPGA. The scaler should apply an effect to a video stream that gives the impression that the video stream is rotated in three dimensions, and create a sense of depth in the video. The scaler should be able to operate in real time on high resolution videos, with a visually pleasing quality. The area of the design should be kept as low as possible."

1.2 Previous work and Contributions

A review of different hardware scaling algorithms have been conducted by Lindø [1].

This thesis continues the work done in the same area during a 15 credits pre project. In the pre project, two different algorithm proposals were discussed and implemented in a high level software model. In the thesis, these algorithms will be iterated on further, and one of them will be chosen for FPGA implementation. The design work will be partially based on an FPGA implemented rectangular scaler, for which the VHDL code is available. This module also has a Matlab model, which was analysed in the pre project.

1.3 Structure

The report is divided into 8 sections, including introduction and conclusion. Sections 2 to 4 introduces previous work that the report benefits from, while sections 5 and 6 discusses the design work carried out during the thesis, and section 7 discusses the verification of the design.

More specifically, section 2 introduces general design decisions for FPGA implementation of image operations, section 3 introduces the algorithms proposed in the pre project, and section 4 contains an analysis of the rectangular image scaler which some of the design work is based on.

Of the design-chapters, section 5 discusses the design and analysis of the model to make it hardware accurate, and also to predict the quality of the different algorithms for hardware implementation. Section 6 contains the actual implementation in VHDL.

Finally, the verification scheme is presented in section 7 and the project is concluded in section 8.

1.4 Relationship with Pre-project

The thesis is closely based on the algorithms proposed in the pre project, and as such, much info from the pre project is required for understanding the thesis. The thesis will not repeat all details from the pre project report. This will mean that some of the mathematical background for the algorithms only will be covered in the pre project report.

The thesis will thus only repeat the main idea of the algorithms from the pre project. It will still attempt to explain the concepts from the pre-project well enough that this report is not needed to grasp the main concepts.

2 Geometrical image operations on FPGA

A geometrical image operation is an operation that changes the arrangements of pixels in an image. This could be magnification, rotation, three-dimensional effects, or more. For a discussion about the different kinds of geometrical transforms, and their mathematical reasoning, see [2].

Bailey divides geometrical scalers into two groups: Forward mapping or reverse mapping. [5] These are suited for different situations, and the choice of which to use should therefore be considered carefully.

2.1 Forward Mapping

In the forward mapping, the position of every input pixel is calculated in the output image. This makes it suitable for streamed input, as every input pixel is only needed once. However, a frame buffer is needed on the output, to store the image while it is being constructed.

A problem with the forward mapping will emerge if the simple approach of assigning the input pixel to the nearest output pixel is attempted. This will lead to holes in the output image, and/or places where one output-pixel is written several times.

To resolve this, it is necessary to map the entire area of the input pixel to the output image. This means that every corner of the image must be mapped, which require large resources.

Bailey [5] discusses several ways to overcome this performance bottleneck by separating the mapping into a horizontal and a vertical part. These methods does however introduce other constraints, and will not be discussed further in this report.

2.2 Reverse Mapping

The alternative to forward mapping is reverse mapping. Here, every output pixel is calculated sequentially, and for each output pixel, the corresponding position in the input image is found. This means that no buffering is required on the output, but the input image must be stored in a frame buffer.

This method avoids the problem with holes in the output, but on the other hand it requires access to sufficient pixels from the input image for every iteration. Exactly how many pixels are required, depends on the interpolation window. To avoid aliasing, a large interpolation window should be selected when the scaling factor is large, i.e. the output image is zoomed out compared to the input.

2.3 Buffer Technology

whichever method is used, a frame buffer is necessary for doing general geometrical transformation, and even if a specialized transformation shall be made, some kind of

buffer is necessary. On an FPGA there are in essence two options of technology for this buffer. It can be implemented with registers on the main area of the FPGA, or it can be implemented on RAM, which would place it on one of the FPGA's dedicated RAM blocks.

For images of larger resolution, however, neither of these options will be large enough. As an alternative, a dedicated RAM block can be used. An external DDR RAM will be large enough to store a frame buffer, and will give a much more efficient implementation of a large buffer, as the RAM units are dedicated for storage. They do not have unlimited bandwidth, however, and access of several pixels from different parts of the buffer every clock cycle may be a problem. The most efficient way to read from the RAM is through burst reads, which makes it possible to read out several successive storage elements on one clock cycle.

One option to overcome the need for bandwidth is to use a two step approach. Have the frame buffer implemented on RAM, but using a cache that stores the pixels that are most likely to be needed again, and fetches new data from the RAM using burst read when necessary. A well implemented cache can reduce the bandwidth of the RAM substantially.

3 Perspective Scaling Algorithms

The pre-project discussed a number of different scaling-related algorithms. The goal of this section is to explain these algorithms well enough that the reader gets an understanding of the good and bad aspects of the algorithms, but not to go in detail regarding the mathematical foundations for the algorihms. For a more in depth analysis, please refer to the pre-project report [3].

The algorithms from the pre-project will be separated in two kinds: *mapping algorithms* and *interpolation algorithms*.

A general image scaler is able to produce an output image where the pixels in the output image represent any set of points on an input image. The *mapping* is defined as the act of finding the position on the input image that each output pixel depends on. This is not generally an exact pixel position, but rather a point in between several pixels. The data structure to hold a specific mapping is a transform. This can be represented in several ways, but the pre-project algorithms uses two different ways which will be discussed later.

The output from the mapper is only a point on the input image. It is the job of the *interpolation* algorithm to convert this point to an actual pixel value. For this job, a filter is required, that calculates the new pixel value based on one or more of the neighbouring pixels to the point to be interpolated.

The following algorithm from the pre-project is modified to emphasise the separation between mapping and interpolation:

Algorithm 1 General Scaling Algorithm

```
for every target pixel p do
  pos ← map(transform, p)
  targetImage(pos) ← interpolate(sourceImage, pos)
end for
```

3.1 Interpolation

Interpolation is generally to find the best line curve through a series of points. In our case, the curve is a two-dimensional function, and the points are the pixels. The operation is also simplified in that we don't need to find the entire curve, but only certain points on it. The points we need to find are the points that shall make out the pixels of the new image.

[3] introduces three different interpolation algorithms, which differ in complexity and quality.

3.1.1 Nearest Neighbour Interpolation

The nearest neighbour interpolation is the simplest form of interpolation imaginable. It is carried out by choosing the closest pixel to the point that is interpolated, and assigning this value to the point. This does obviously not give a very good approximation to the true value, but due to the extreme simplicity of the algorithm, both in terms of resources and implementation, it is still a popular choice [2].

3.1.2 Bilinear Interpolation

The next step up in complexity is the bilinear algorithm. It calculates the intermediate value between four neighbouring pixels, and requires three multiplications.

This algorithm calculates the weighted average of the four surrounding pixels to the point to be interpolated. The weighting is the distance between the point and the pixel borders, so that the pixels closest to the point has the most influence on the calculated interpolation value.

Below you can see the Matlab implementation of the algorithm from [3]. Note that this code does not handle picture borders. This has been left out because it is not essential for understanding the code, as the special cases it requires are fairly obvious. By adding them, however, the readability of the code suffers. To see the full Matlab implementation, please see appendix **??**.

```
function value = interpolatePoint(x,y,sourcePic)
  u = floor(x);
  v = floor(y);
  a = x-u;
  b = y-v;
  A = sourcePic(u,v);
  B = sourcePic(u+1,v);
  C = sourcePic(u,v+1);
  D = sourcePic(u+1,v+1);
  E = A + a*(B-A);
  F = C + a*(D-C);
  value = E + b*(F-E);
```

3.1.3 Polyphase filtering

Polyphase filtering is the last interpolation algorithm discussed in [3]. It is a more complex and general algorithm. It uses a coefficient matrix to store p phases of an interpolation kernel. Each value of the kernel codes the weighting of one pixel surrounding the interpolation point. The interpolation value is thus found by first selecting an appropriate phase, depending on the distance of the interpolation point to the middle of the surrounding pixel. When a phase is selected, the corresponding coefficient matrix indices (which contains the weighting of the pixel) are multiplied with the surrounding

Table 1: Overview over how many pixels are used to calculate the interpolated value for the different interpolation algorithms. For the polyphase filter, there is no strict limit for this, and it is commonly seen everywhere from 16 to 256 pixels

Algorithm	Pixels
Nearest Neighbour	1
Bilinear Interpolation	4
Polyphase filter	> 4

pixels, and the result is added to get the interpolation value. The filter is said to have n taps if it calculates the pixel value based on $n \cdot n$ surrounding pixels.

The algorithm can be summarized in the following high-level algorithm:

Algorithm 2 Polyphase Filtering Algorithm

$P_i \leftarrow$ interpolation point
$\theta \leftarrow$ getPhase$(P_i - floor(P_i))$
$C \leftarrow$ getCoeffSet(θ)
$P \leftarrow n \cdot n$ pixels surrounding P_i
for all p in P **do**
  $sum \leftarrow sum + p \cdot C_i$
end for

The quality of the polyphase filter naturally depends on the coefficient matrix. This is usually some kind of approximation to a sinc.

3.1.4 Choosing an Interpolation Algorithm

When choosing which algorithm to use, it is obviously important to consider the trade off between quality and complexity. But that is not the only thing that is important to consider. The different algorithms gives different scaling artifacts, like blurring or aliasing. When downscaling using large scaling factors, many filter taps is required to avoid loosing data in the scaled image.

3.2 Mapping Algorithms

3.2.1 Matrix Mapping

The matrix mapping algorithm is developed by [2] and adapted to Matlab by [3]. The algorithm uses a reverse mapping, and is the most common way to do geometric transformations of images in software. By carrying out a matrix multiplication between the pixel coordinates and a transformation matrix, a new set of transformed indexes are obtained. These new indexes can then be sent through an interpolation filter to obtain new pixel values.

It is possible to represent the transformation with different matrices, depending on the required generality. With a 2-by-2 matrix, all affine operations can be represented.

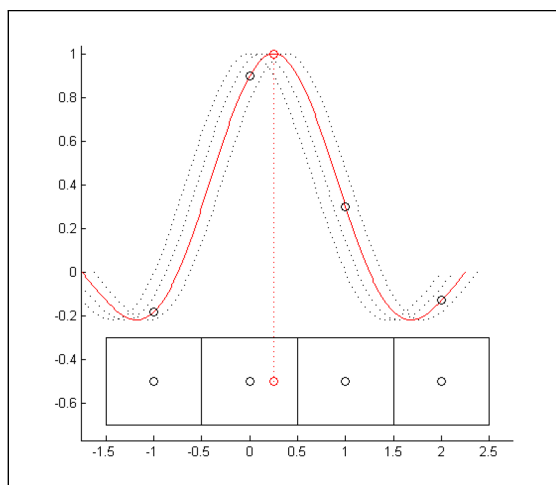


Figure 1: Illustration of the different phases of a 1D 4-phase polyphase filter with 4 taps, with a row of pixels illustrated below. The filter has four sets of coefficient functions (the dotted curves). For each interpolation point, one of these are selected. In the example, an interpolation point is given (red circle), and the appropriate phase is selected (the red curve). The black circles shows what values of the function that needs to be stored, and that are multiplied with the pixel values

By increasing the size of the matrix, more transformations can be represented. For a perspective operation, a 3-by-3 matrix is required. The matrix product in that case would look like equation 1.

$$\begin{pmatrix} h'x' \\ h'y' \\ h' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{1}$$

In normal mathematical operations, this would write out as equations 2 and 3.

$$x = \frac{a_{11}x_0 + a_{12}y_0 + a_{13}}{a_{31}x_0 + a_{32}y_0 + a_{33}} \tag{2}$$

$$y = \frac{a_{21}x_0 + a_{22}y_0 + a_{23}}{a_{31}x_0 + a_{32}y_0 + a_{33}} \tag{3}$$

To find the matrix for the desired transformation can be complicated with this algorithm. Due to this, a script has been created in the pre-project to calculate the matrix from the angle of rotation of the picture compared to the observer, and the relation between the distance to the observer and the size of the picture. For details about this algorithm, and the underlying mathematics, see the pre-project report [3].

The matrix algorithm is a general algorithm able to do a wide range of transforms in addition to perspective scaling. This generality, however, comes at a cost of higher complexity. As can be seen from the equations above, the calculations required for each index are quite complex.

3.2.2 Iterative Mapping

The *reference based algorithm* was developed in the pre-project as a less resource intensive alternative to the matrix algorithm [3]. In this report, the algorithm will be renamed the *iterative algorithm*, to better capture the behaviour of the algorithm. In this section, the work on the algorithm that was carried out in the pre-project will be introduced, and in later sections, the algorithm will be further developed, and adapted for hardware implementation.

The core idea of the iterative algorithm is to not calculate every single interpolation point, but rather their relation with each other. The algorithm is iterating over the input image, and calculating the distance to the next position, instead of the whole position. This means that this algorithm also employs a reverse mapping.

To explain the algorithm, let us imagine what is intuitively necessary for making a perspective scaled image. To illustrate, we will use the example given in the pre-project of a rotatable picture viewed from a camera, as shown in figure 2.

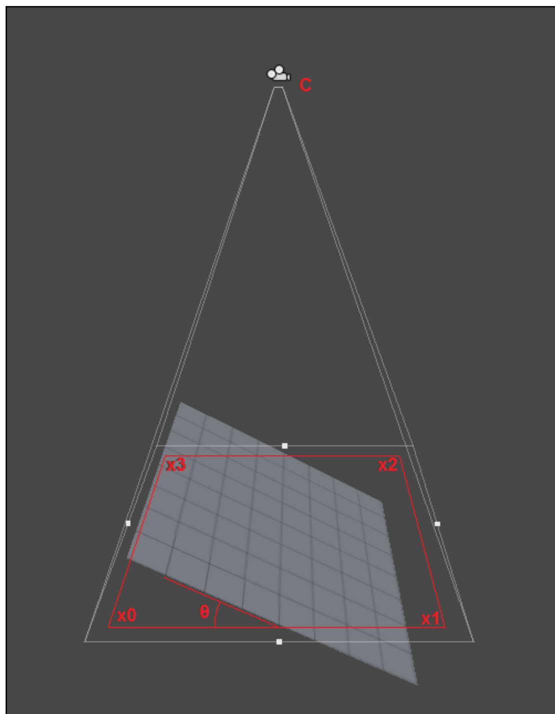


Figure 2: A visualization of a picture, viewed from a camera at the top, rotated by an angle θ compared to the original position. The original position is marked by the red frame, and the view rect of the camera is given by the white lines outside of the red square. Figure form [3].

Firstly, the picture will look slimmer, as it will be seen partially from the side. Therefore, a scaling of the picture in the horizontal direction is necessary.

Secondly, the left part of the picture will be closer to the camera, so it will need to be zoomed in, and the other side of the picture will need to be zoomed out. We still want the picture to fit inside the camera bounds, however, so the entire scene will need to be zoomed out by a factor sufficiently large for the left edge of the picture to fit inside the

image. This is equivalent to only zooming the right part of the image out, and leaving the left part as it is.

We will separate the operation into horizontal and vertical scaling, as they work slightly different:

In the horizontal direction, the image should be scaled with a scaling factor that increases linearly from left to right in the image, and the width of the image should be reduced. By initiating the horizontal scaler with scaling factor 1 on the left edge of the image, and increasing it by a constant amount for each pixel to the right of the image, this will be achieved.

The scaling operation would be a bit different in the vertical direction, where the output image is not square. The height of the columns in the image should be reduced for each step to the left of the image. This would leave a triangular area above and below the image with no data. The horizontal scaling would also leave an area to the right of the image without data. There are many ways of handling these areas, but the simplest one is to fill them with a desired background colour. In this project, black will be used.

4 The Polyphase Scaler

This section will introduce a state of the art rectangular image scaler that is implemented on FPGA. The scaler will be referred to as the *polyphase scaler* in this report, from the interpolation algorithm it uses. The scaler is responsible for converting video streams to different resolutions in real time, and it is able to operate on high definition streams, with any rational scaling factor.

Even though the polyphase scaler is only able to convert a rectangle to a rectangle, there are still some parts of the design that can be used in a general perspective scaler, and this section will attempt to identify those parts.

4.1 Algorithm

The core of the scaler is a polyphase filter, which is described in section 3. The filter can change between three different modes: 4, 8 and 16 taps filtering, which means that it uses an filtering window between 16 and 256 pixels large. The scaler is able to switch between the different modes based on the desired magnification. For this job, three different sets of coefficients are necessary. The coefficients are stored in memory, and contains 32 phases for each of the modes.

4.1.1 Separation of Filter Operation

The scaler is calculating the value of one output pixel each clock cycle. This means that all filter operations needs to be done in parallel, which requires large hardware resources. A normal polyphase filter would require n^2 multiplications pr pixel, where n is the number of taps in the filter. In this scaler, however, the filtering operation is separated into a horizontal and a vertical part, which reduces the number of multiplications to $2n$. Figure 3 shows how the filter is built up, along with the buffering that is required.

It is possible to think of the filter as a combination of a n-by-1 filter and a 1-by-n filter. But this architecture is not equivalent to a n-by-n filter in all cases. We will now take a closer look at the assumptions that this architecture is built on. Figure 4 shows how five operations with this filter combines to do one n-by-n operation. These five operations requires a total of n^2+n multiplications. The savings of operations happen because many of the 1-D scaling operations can be reused, and in average, only one 1-D operation is required in each direction, per pixel.

By studying figure 3 and 4, we can find the assumptions that this filtering algorithm builds on. Firstly, we see that the filter must operate in a left-to-right, or other orderly fashion. It is not possible to choose which pixels to interpolate in an arbitrary fashion, as the vertical scaler needs the pixels in a given order.

Secondly, the magnification needs to be identical over the entire picture. The algorithm depends on using one horizontal interpolation several times, but if the magnification was changing, the previously calculated values would be wrong when you moved further left in the image. If these needs to be recalculated, all the savings of this algorithm are

lost. In essence, this means that the filtering algorithm is only suitable for rectangular output, and it would be difficult to reuse for a perspective scaler.

4.2 System Architecture

In order for the scaler to be able to operate on a video stream, as one part of a video processing unit, a large system is required. A complete analysis of this system is beyond the scope of this report, but instead, a simplified overview of the parts of the system most closely related to the scaler will be given, and some hints will be given to other parts of the system.

4.2.1 Scaling Colours

The scaler that has been discussed so far, only scales a grayscale-image. To be able to scale a coloured video stream, one grayscale-scaler is used for each colour component. The system uses 4:2:0 sub-sampled Chroma components. This means that there is time to calculate both chroma components on a single scaler, and only two core scalers are required.

4.2.2 Rate Control and Buffers

An interface standard from Altera called Avalon Streaming is used for communication between the modules [4]. This standard includes a backpressure system, which enables a sink module (defined as a module that is receiving data from another module) to control

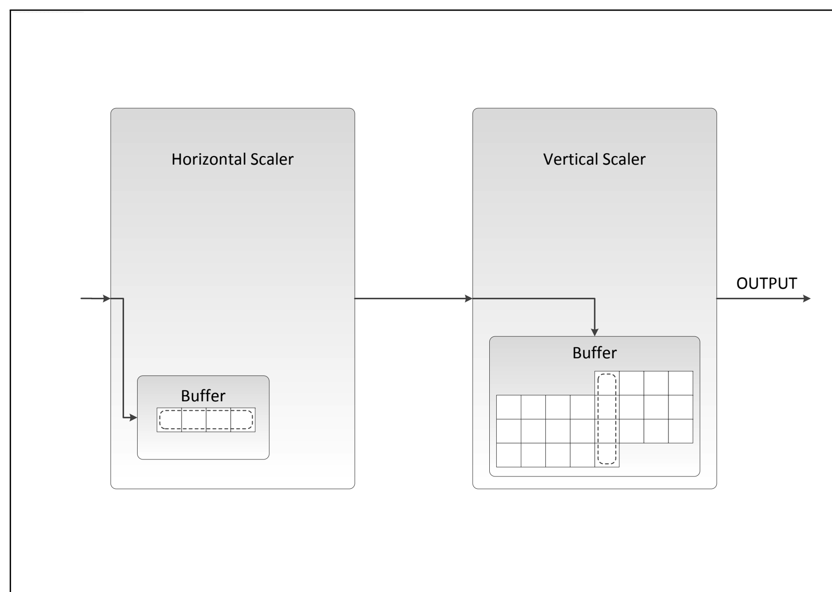


Figure 3: The separation of the filter into a horizontal and a vertical part, and the buffers required in each case. The dashed circle shows the pixels that are used by the filter for one operation (the filtering window).

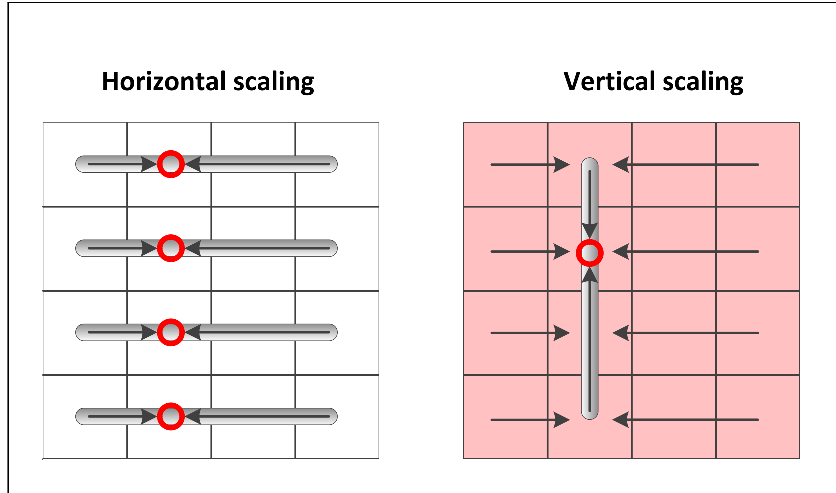


Figure 4: Shows the five 4-by-1 filtering operations that corresponds to one 4-by-4 filtering. The red pixels are scaled horizontally, and each of them ”contains” the result of an interpolation of four pixels (symbolized with the arrows). When they are scaled by the vertical scaler, the resulting operation is equivalent with a 4-by-4 interpolation with coefficients equal to the multiplication of the x- and y-coefficient of the 1-D interpolation. Note that the 1-D operations are not carried out in this order.

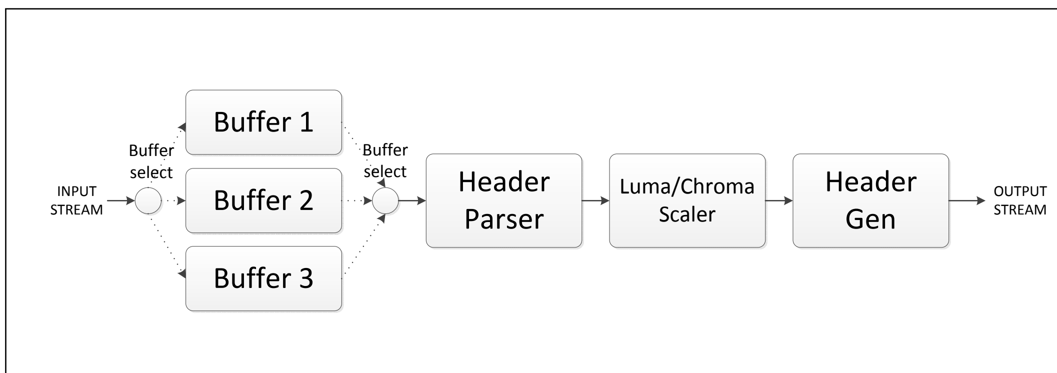


Figure 5: Simplified system architecture for the polyphase scaler.

the rate of its input by setting a ready-signal that is sent to the source module. When this ready-signal is low, the source-module ceases calculations of new values, and the last value is kept on the output until ready is high. This means that the output modules decides the data rate of the system.

The system is able to handle different frame rates on the input and output. This is made possible by a triple buffer system (see figure 5). At any one time, one buffer is locked by the write unit, one is locked by the read-unit, and one is free (the states are not shown in the figure). The write-unit is receiving the video stream from the input, and writes it to the buffer at the rate chosen by the input stream. Similarly, the read-unit reads from the read-buffer at the rate chosen by the output frame rate.

When the write-buffer is finished with a buffer, the lock is released, and the free buffer is locked, and made the new write-buffer. The read-unit works in much the same way, except when it is done, the most recently written buffer is chosen to be the new read-buffer (this could be either the same buffer, or the free buffer).

This behaviour will cause frames to be repeated on the output if the output rate is higher than the input rate, and frames to be skipped if the input rate is higher than the output rate. It will also make sure that there always is a buffer ready for reading or writing, so that no data is lost, and there always is data on the output. If the rate is higher on the input than on the output, full frames are discarded, which causes the least visual impact.

4.2.3 Headers and Video Stream Handling

The scaler system also has modules for converting the video stream between what is used internally in the hardware, and the format used on screen, and header parsers and generators to remove and regenerate the headers from each frame. These will not be discussed further in this report.

4.3 Relevancy for Perspective Scaler

The motivation behind the analysis of the polyphase scaler was twofold. Firstly, it was meant as an inspiration for the implementation of the perspective scaler. By studying a similar design, some pointers could be had to how the perspective scaler should be designed. Secondly, the goal was to decide which parts of the polyphase scaler could be reused in the perspective scaler, and which that required redesign. Indeed, the analysis has shown that designing a full scaler system is no small undertaking, but it has also shown that this is not necessary. The system surrounding the scaler will behave identically whether a rectangular scaler or a perspective scaler sits inside of it, as long as it sends out a video stream of rectangular images in the end. This means that most of the system can be reused for a perspective scaler design. In section 6, this will be further discussed.

5 Perspective Scaler Model

This section will present the work carried out to design software models for a few of the most promising algorithms for hardware implementation. The motivation for a software model is two-fold: Firstly, it functions as a cheap prototype, allowing the performance of the algorithm to be assessed before the bulk of the implementation work is done. Secondly it will be used to test the hardware unit once it is designed, to compare the output from the two.

Many possibilities were considered when choosing an algorithm for the software model, but the choice was made to focus on the algorithms introduced in the pre-project. These algorithms are shown to be working, and they are well known, reducing the cost of implementation.

The relevant algorithms are introduced in section 3, but this section will take hardware considerations into account, and introduce some changes.

5.1 Modular Architecture

To make the hardware unit as modifiable as possible, where the different algorithms can be changed between easily, a modular architecture has been chosen for the scaler. The architecture is inspired from that of the matrix implementation of the pre-project, and it presents a similar interface, making the different algorithms easily interchangeable and easy to compare. This architecture will separate the implementation of the interpolation and the mapping, so that the interpolation algorithm can be interchanged without affecting the mapping algorithm, and vice versa.

As discussed in section 4, the filter from the polyphase scaler is not compatible with the complex order of interpolation required for perspective scaling. Instead, an ordinary filter, without separation into a horizontal and a vertical part will be used for the iterative algorithm. The extra hardware resources associated with this can be compensated for by choosing a simpler algorithm.

The following matlab code uses two functions, interpolatePoint and mapPoint, to implement the perspective scaling algorithm. The in-parameter *mappingData* is a placeholder for the data that the mapping algorithm needs to carry out the mapping, and varies with the algorithm.

```
function picOut = transformPic(picIn, mappingData)
  [height width] = size(picIn);

  % possible modification of trasformData...

  picOut = zeros(height, width);
  for u = 1:width
    for v = 1:height
    [x y] = mapPoint(v,u,mappingData);
      picOut(u,v) = interpolatePoint(x,y,picIn);
    end
  end
```

The pre-project compares the two different mapping algorithms, and concludes that both have merit, even though the iterative algorithm shows much better potential for a low-resource implementation. The matrix algorithm is well modelled in the pre-project, and is ready for hardware implementation, given that a suitable implementation for the multiplication and division circuits could be found.

The focus here will therefore be put on the iterative algorithm instead. The iterative model from the pre-project is not very hardware accurate. Therefore, a complete redesign has been carried out, but with the idea behind the algorithm remaining the same.

The matrix algorithm will rather be used as a gold standard to compare the iterative algorithm against, as this algorithm is known to be producing an ideal mapping.

5.2 Iterative Mapping Model

The iterative algorithm was created based on the assumption that the filter from the polyphase scaler could be used. This has been shown not to be the case. In addition, it is not clear how all aspects of the algorithm should be implemented in hardware. The algorithm will therefore be redesigned with the new separation of filter and mapping, as well as hardware implementation, in mind.

The algorithm is still split into two parts, one horizontal and one vertical, and the idea to increase the scaling factor linearly from right to left in the image remains the same. The way this is done is a bit different from the algorithm in the pre-project, however.

First, let us discuss what the *scaling factor* means. In the polyphase scaler, the scaling factor is the size of the input image divided by the size of the output image. That means that a large factor means the image will be scaled down. The polyphase scaler has one scaling factor for each dimension; horizontal and vertical.

This scaling factor relates to the steps the filter is taking in the input image. With a scaling factor of two, the filter is moving two pixels on the input image for every pixel on the output image, resulting in the output image size beeing halved. Small steps in the input image will in turn lead to magnification.

To do perspective scaling, we want to zoom in the left part of the image, and zoom out the right part. That means that we want to start out with a small scaling factor, and

increase it the further right we get.

5.2.1 Horizontal Scaling

In the horizontal direction, a register is used to store the value that the scaling factor increases with, and this is added to the register holding the actual scaling factor for every step right in the image. The scaling factor is again added to the register holding the filter attack point, just as in the polyphase scaler. The architecture is shown in figure 6.

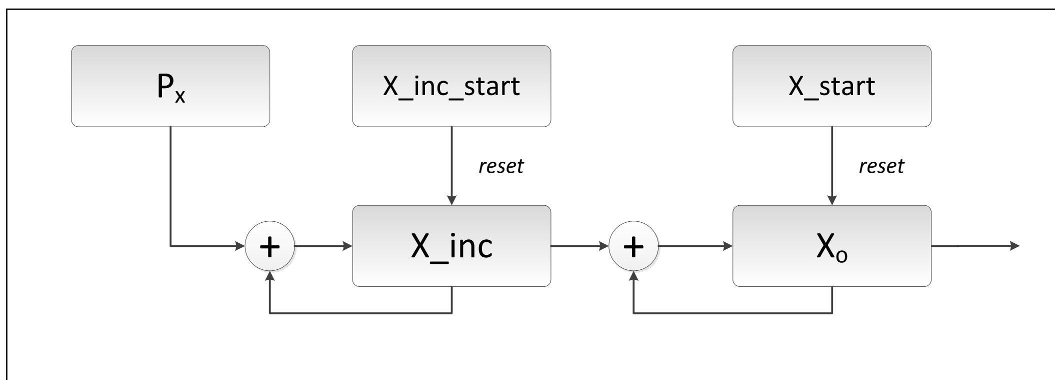


Figure 6: Architecture for perspective mapping calculations in the horizontal dimension

5.2.2 Vertical Scaling

In the vertical direction, we don't want the scaling factor increased for every vertical step in the image, as would be equivalent with the horizontal architecture. Rather, we want the vertical scaling factor to increase for each step horizontally. The way we will do this is to think about the rows in the image as lines, and modify the slope of those lines.

It is tempting to add this slope to Y_0 for every step, thinking that the x-step is one, so the slope would be equal to the vertical step size. If the horizontal scaler took steps of 1 to the right in the image, this would be true, and a straight line could be made by just adding the slope to Y_0 for every step. But since the step size (x_inc) of the horizontal scaler varies, the slope needs to be multiplied with x_inc to get the correct step increase in the vertical direction.

To have a choice of algorithm that does not require multiplications, however, both version of the algorithm will be modelled. The algorithms will be called the simple and the improved iterative algorithm. In the last part of this section the performance impact of this weakness will be investigated.

The architecture for the simple iterate algorithm is shown in figure 7. In the improved iterate algorithm, the slope would need to be multiplied with x_inc from the horizontal calculations before it is added to Y_o.

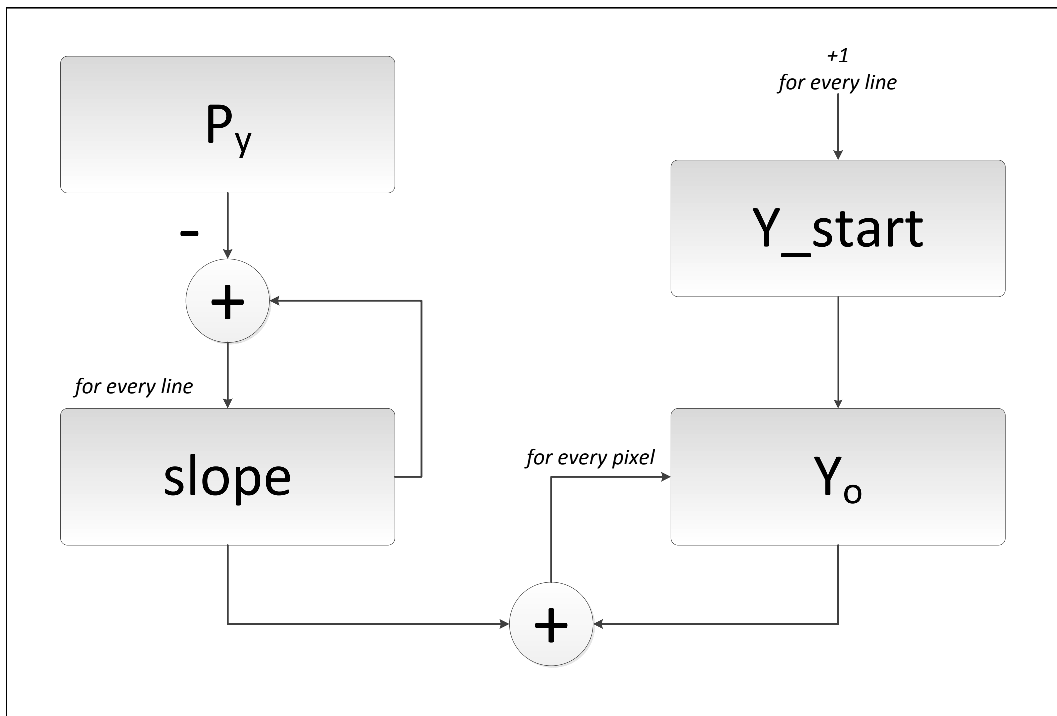


Figure 7: Architecture for the perspective mapping calculations in the vertical direction

5.2.3 Matlab Implementation

Below, the code from the Matlab implementation is given. The matlab model uses a mathematical formula instead of registers to calcualte the values, but two ways of doing it are equivalent. Some variables that are calculated in the matlab model, however, will be stored in RAM in hardware to reduce complexity.

```
function [xo, yo] = getPerspectiveMapping(xi, yi, dim, P)

if (nargin < 4)
  P = [1.6720 0.6080];
end

Px = P(1); Py = P(2); % Perspective factors for x and y dimension
H = dim(1); W = dim(2); % height and width

xo = 0.5 + xi*(1+0.5*xi*(xi+1)*Px/W^2);
slope = Py*(H-1)/(2*H) - Py*yi/H; % slope start point will be stored
    in RAM
%Choose one version:
%% Simple Version: %%
yo = 0.5-slope*xi+yi; % '-' because index 0 is at the botton of the
    image.

%% Improved Version: %%
% yo = 0.5-slope*xo+yi; % using xo instead of xi is equivalent to
% multiplying slope with x_inc
```

The variables Px and Py are responsible for controlling the amount of perspective applied to the images. Larger values will give more perspective scaling, but the ratio between Px and Py should be chosen carefully, as this will influence the quality of the scaling operation.

The current values of $Px = 1.672$ and $Py = 0.608$ are found by trial and error, and are optimized for the improved iterative algorithm.

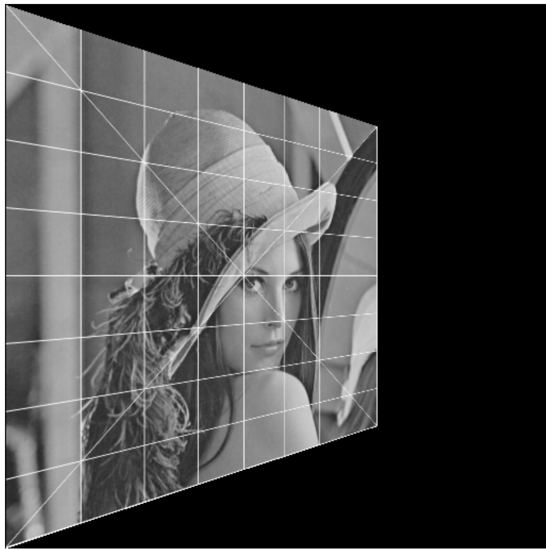
5.3 Quality Comparison

The quality comparison of the images from the models will guide the choice of algorithms for the hardware implementation. The matrix algorithm will be used as a gold standard. The two iterative mappers will be compared to this, and errors will be discussed.

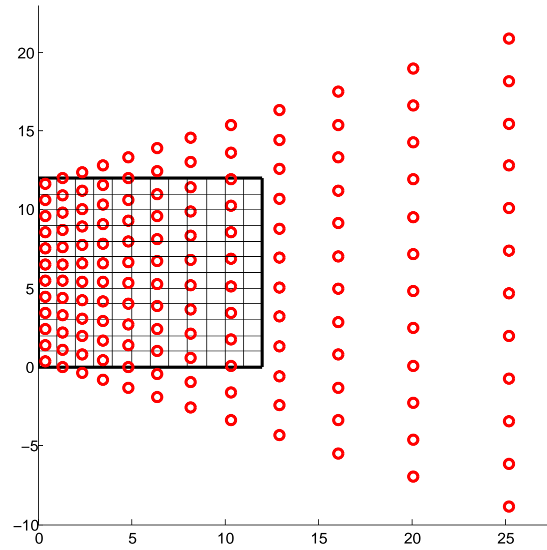
Additionally, bilinear and nearest neighbour interpolation algorithm will be compared. The bilinear implementation exists from the pre-project, while the nearest neighbour has been implemented for the thesis. Its implementation is so simple that it will not be discussed.

The test images for the mapping algorithms are paired up with an image showing the mapping of the respective algorithms. These are created by running the mapping algorithms on an imaginary 12x12 image. But instead of interpolating the positions, they are plotted in a graph as circles, representing the interpolation point sent to the filter. The black wire-frame is added to represent the input image. This has proved a good way to investigate errors and differences in the mapping algorithms during the work with the thesis.

Figure 9 shows the simple iterate algorithm, and is showing a clear artefact caused by the error with the vertical scaling. Compared to the problems with the iterate algorithm in

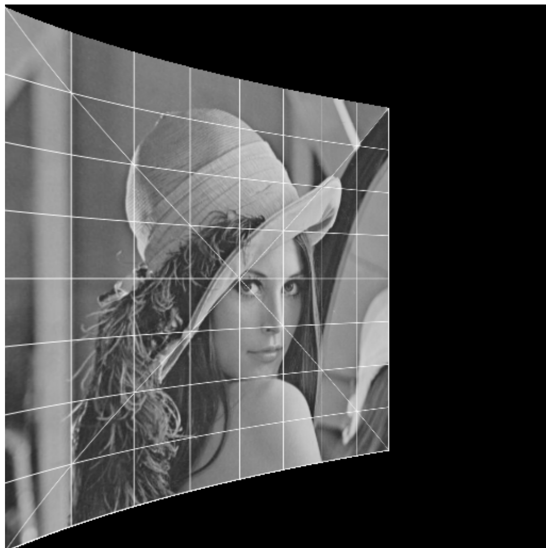


(a) Matrix test image

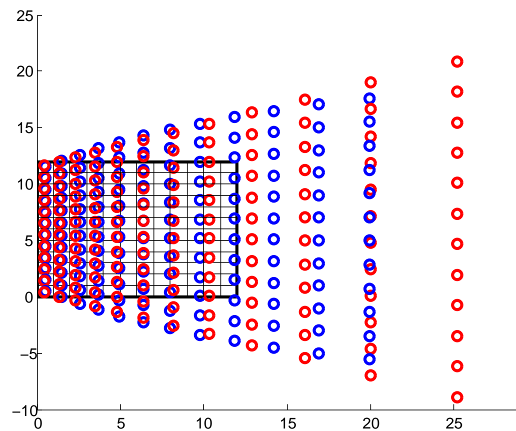


(b) Matrix mapping

Figure 8: The matrix mapping algorithm that is used as a gold standard for the visual comparison. a) shows a test image scaled with the matrix mapping algorithm and the bilinear interpolation algorithm, and b) shows a visualization of the mapping of the matrix algorithm. The black wire-frame represents a 12x12 pixel input image, and the circles represent the pixel positions of the output image as given by the mapper (the interpolation points, in other words). The circles that are outside the borders of the input image makes out the black part of the image.

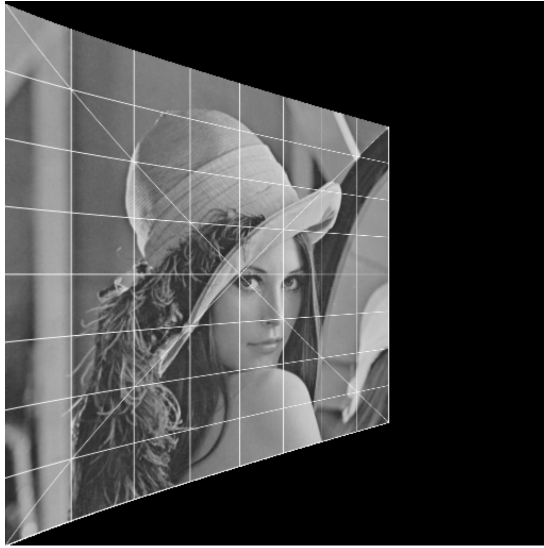


(a) Simple iterate test image

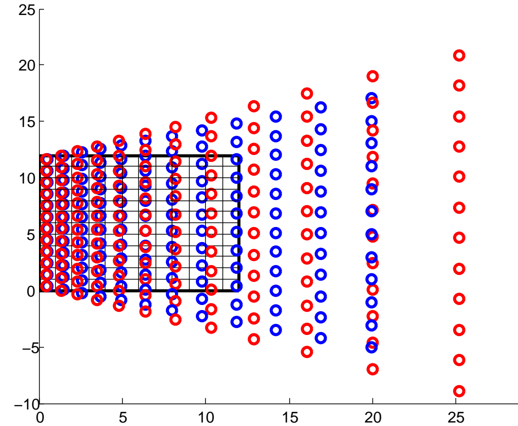


(b) Simple iterate mapping

Figure 9: The simple iterated algorithm, without multiplications. a) shows a test image scaled with the algorithm along with a bilinear interpolation filter, b) shows the mapping of the algorithm (blue) compared to the matrix algorithm (red). Notice how the rows in the mapping is bending in towards the middle, resulting in a stretched look in the image.



(a) Improved iterate test image



(b) Improved iterate mapping

Figure 10: The improved iterated algorithm. a) shows a test image scaled with the algorithm along with a bilinear interpolation filter, b) shows the mapping of the algorithm (blue) compared to the matrix algorithm (red). The error in the vertical part is corrected, but the change in the scaling factor (the distance between the circles) in the x-direction does not match the matrix algorithm completely.

the pre-project that the diagonal lines were bending, this error is much more noticeable. The positive development, compared to the pre-project, is that the mapping view makes it much easier to investigate the errors, and may be used to correct the error more easily. This is a benefit gained from the modular architecture and interface reduced in the redesign of the algorithm.

In the improved iterate algorithm (figure 10), the vertical error is resolved, resulting in a much more pleasing scaling result. By inspecting the image carefully, however, a small error can still be found. This is most noticeable towards the left edge of the image. By studying the mapping comparison, we see that the scaling factor changes too little in the beginning and too much in the end, compared to the matrix algorithm. This means that the assumption that the scaling factor was linearly dependent seems to be wrong. But all in all the improved iterate algorithm gives a good approximation to the matrix algorithm, for a large reduction in complexity.

Lastly, figure 11 compares the bilinear interpolation algorithm, that has been used on all images up to this point, with the simpler nearest neighbour algorithm. As expected, the nearest neighbour algorithm is showing some artefacts, but the quality is good enough to be suitable for a prototype module, for instance.

(a) Bilinear algorithm

(b) Nearest neighbour algorithm

Figure 11: A comparison between the bilinear and the nearest neighbour algorithm. Both images are scaled with the matrix mapping algorithm.

6 Hardware Implementation

Section 4 concludes that the system used for the polyphase scaler can be adapted, without too many changes, to fit a perspective scaler instead. This system will therefore be the starting point for the design of the perspective scaler.

The top design unit of this project will take over the role of the core, single colour component, scaler in the polyphase scaler system. To reduce the amount of redesign required, the interface of the perspective scaler will be chosen similar to the polyphase scaler. It can not be made identical, however, as, for instance, the perspective scaler needs a few extra variables, defining the perspective transforms. This means that some system redesign is necessary. The thesis, however, will only focus on the core scaler, which means that some further work will need to be carried out if the module is to be put into use.

Based on the model developed in the previous section, this section will discuss the actual hardware implementation of the module. To make the module executable without the system, a crude frame buffer will be designed, and implemented as on chip RAM. Later, in section 7, the module will be simulated and verified.

The section will start out by discussing the framework from the polyphase scaler, and later, the architecture introduced in section 5 will be used as a guide for the design of the core perspective scaler. Finally, an analysis of the designs architecture will be presented, which will serve as a reading guide and documentation for the VHDL code.

6.1 Framework

Even though no work has been put into design of the frame work for this project, a small discussion of how the different parts are fitting with the system will be given.

The part of the polyphase system that is responsible for stream conversion and header parsing/generation can be readily reused for this scaler. This is also the case for control logic for operating the different colour components and use of the chroma scaler. The fact that the core scaling operation is a perspective scaling does not change the functionality of these modules at all.

The register interface of the system, responsible for storing data from software, needs to be updated, however. New variables controlling the perspective scaling operation are needed, and these needs to be written from software. This means that some registers will need to be added, and new addresses made available for software.

The system also includes a frame buffer. In theory, this could be redesigned to replace the buffer used in the perspective scaler design, but this would require system redesign.

6.2 Core Perspective Scaler Design

The core scaler is divided into three parts (see figure 12). This division implements the modular architecture discussed in section 5.

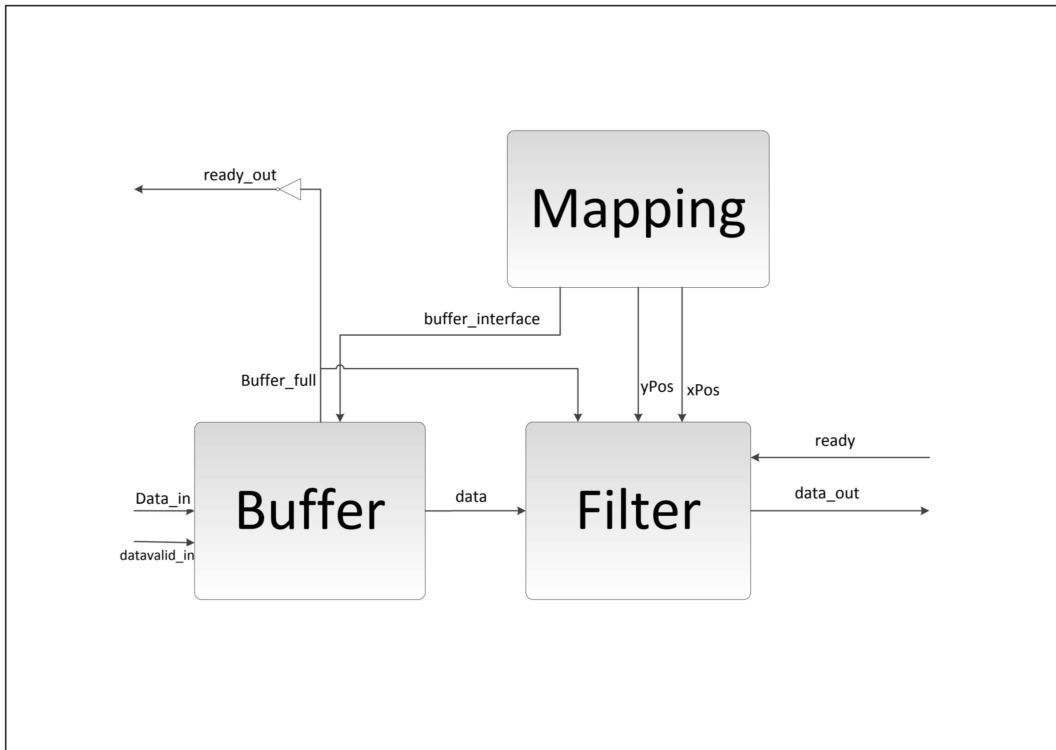


Figure 12: High level block diagram and interface of the hardware unit.

The motivation for the modular architecture was to allow the different parts of the design to be replaced, while keeping the effect to the rest of the scaler as small as possible. The most challenging part of the scaler to design in a modular way has turned out to be the buffer, and the link between the buffer and the filter. This is because the filter (depending on algorithm) is depending on an unspecified number of pixels from the input frame.

This means that when choosing a filter, the specifications for the buffer is also chosen. The nearest neighbour filter requires only a single pixel value from the frame buffer for each clock cycle, while, for instance the bilinear, requires four pixels every clock cycle. This would therefore require some kind of cache.

To keep this implementation simple and to the point, the simplest filtering algorithm has been selected. With the nearest neighbour filtering algorithm, the frame buffer can be simulated with a buffer on one of the on-chip RAM blocks, and it does not need a cache in order to reduce bandwidth.

The choice of interpolation algorithm and buffer solution have no implication on the choice of mapping algorithm. They should all work with the current configuration. For this implementation, the simple iterative algorithm has been chosen. As the iterative algorithm is a new algorithm introduced in this report, there are no other implementations available to guide future work.

Optimally, several of the algorithms should have been designed and compared, but time did not allow for this. Therefore, the simple version of the iterative algorithm was chosen,

as this is the fastest to design.

6.3 Architecture Documentation

The top level of the design defines the interface between the sub-modules, for which a simplified illustration is given in figure 12.

The iterative mapper is built up of several register controlling the variables shown in figure 6 and 7. For every one of these variables, a range must be selected. This range is defined by the bit-length of the register, and the defined decimal point. This point is not relevant for the specific register, and there is no way to see it from the register, but is defined, and used when operations are done on several registers (like adding one to another).

Figure 13 lists all the registers used in the mapper implementation, and the relation between them

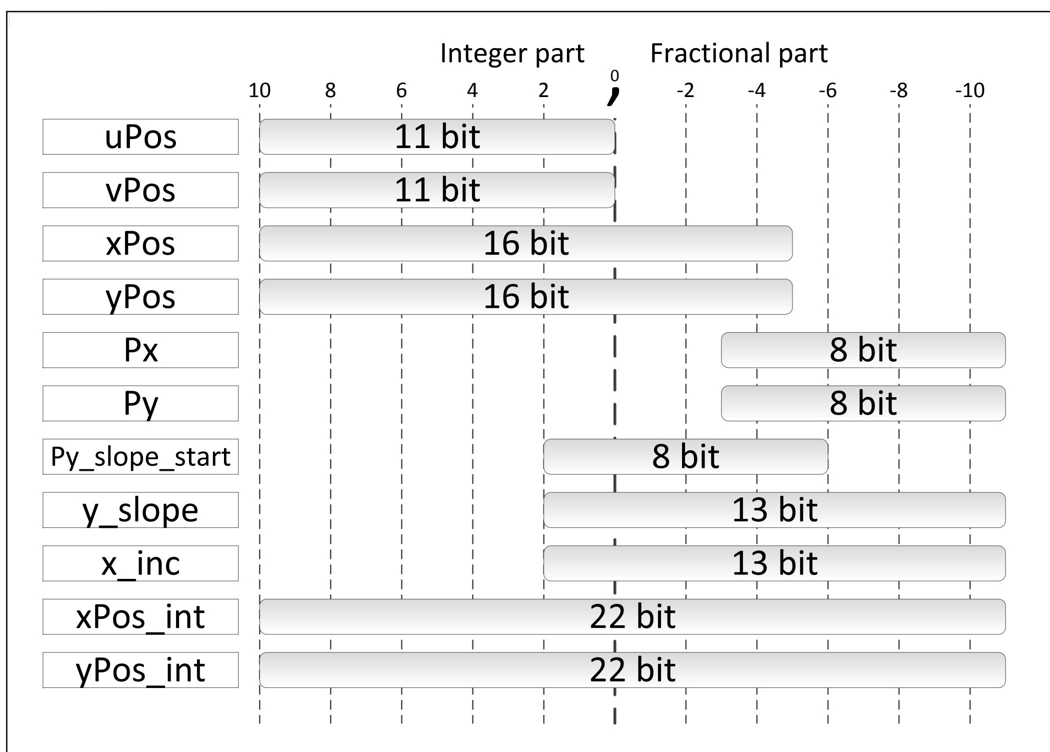


Figure 13: Overview over the range and bit length of all mapper variables.

As input, the system takes two signals to define the size of the input: *input_vsize* and *input_hsize*. The current implementation does not support different sizes on input and output, so these signals will also define the output. There are three variables that define the perspective transformation: *Px*, *Py* and *Py_slope_start*. Px and Py are known from the Matlab model, but they have a slightly different functionality here. The software model calculations are dependent on the video resolution. This causes unnecessary complexity in hardware, and this dependency is therefore baked into the signals. The signal *Py_slope_start* is given by Py and input height. It is used to initialize *y_slope* so

that its value goes from Py_slope_start to $-Py_slope_start$ over the frame. Equation 4 defines the relation between the software and the hardware variables.

$$\begin{aligned} Px_{hw} &= \frac{Px_{sw}}{W^2} \\ Py_{hw} &= \frac{Py_{sw}}{H} \\ Py_slope_start &= \frac{1}{2} Py_{hw}(H-1) \end{aligned} \tag{4}$$

6.4 Potential for improvement

The implementation chosen for this thesis is very simple, and as such, not the best visual quality is expected. In order to improve this at the cost of complexity and FPGA, the simple algorithms can be substituted with somewhat more complex ones. The iterative algorithm can be redesigned into the improved version, whose architecture is discussed previously. Additionally, improvements will be suggested for the filtering and buffering modules.

6.4.1 Filtering

The substitution of the nearest neighbour filter with a bilinear filter would go a long way in improving the quality of the scaler, at a reasonable cost in hardware resources. The filter can be implemented by three multipliers and three adders, as shown in figure 14.

The bilinear filter requires four pixels from the input image, which will require a more capable buffer solution, compared to the nearest neighbour filter that only needed one pixel.

6.4.2 Buffering and Caching

The polyphase scaler uses a frame buffer to convert between different frame rates, and it would be advantageous to use this to do most of the buffering job. This would require a redesign of some of the system, however, and could therefore require a bit of work.

The buffer is implemented on external RAM, and it is inefficient to read out many single values from this ram. It is more efficient to use burst reads to read several neighbouring values at once. If this is combined with a cache, a quite bandwidth efficient solution can be made.

A cache can be made very complex or very simple, depending on how much intelligence is implemented for choosing which pixels to keep, read and overwrite. When designing a cache for the perspective scaler, the access pattern of the input frame should be taken into consideration. The mapping illustrations from the last section (for instance figure 8b) gives an idea of how this looks. The next pixel needed is usually located on the same line as the current pixel, and when its not, its usually located on a neighbouring

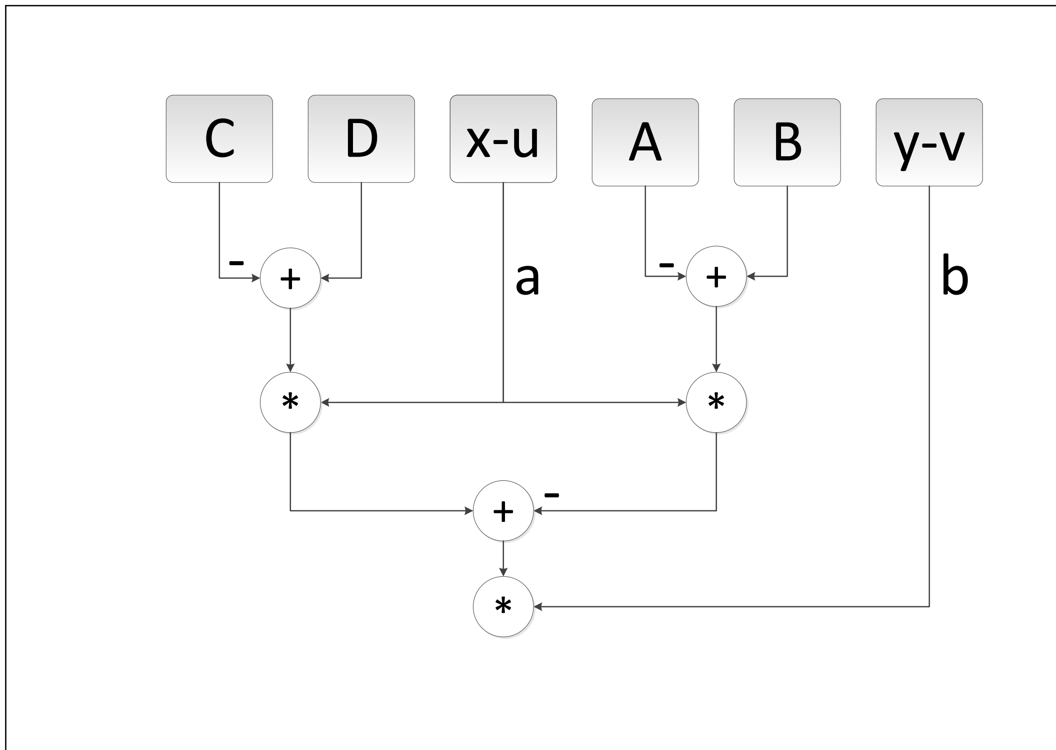


Figure 14: Block diagram for the bilinear architecture

line. The only exception to this is the beginning of a new line, where the location on the input frame corresponds to the location on the output frame.

Putting this knowledge together, a high-level caching algorithm can be suggested. The cache should store a number of lines that is larger than the interpolation window (for the bilinear algorithm, this means at least three lines). When the filtering window is approaching the edge of the cache, the unused line should be discarded, and the line that is needed next should be read in to replace it. In the top half of the image, the cache should always read the line above the current, and oppositely in the bottom half of the image.

Additionally, the cache should have some logic to remember how a new line starts. It will have time to set up to this during the period at the end of the line when the mapping location is out of bounds.

7 Performance Analysis and Verification

This section will discuss the testing and synthesis statistics of the hardware module discussed in the previous section. Firstly the verification scheme for the module will be introduced and discussed. Later the synthesis report from Quartus will be analysed, and some predictions will be made to how the size varies with resolution.

7.1 Verification Plan

7.1.1 Scope

To test a hardware system, several tests are usually developed, for each of the different modules of the design. In addition, high level tests should be used.

Testing the single perspective scaler module is luckily a simpler task. As discussed in section 6 the goal of the design is not to create a module that is ready for shipping, but only a prototype to prove that the algorithms are working, and to test them on actual hardware. This will influence the testing of the module. The test will not put emphasis on details of the timing requirements and communication protocols that are important when communicating with surrounding modules. Rather, the core functionality of the scaler is in focus: The correct perspective scaling of images.

7.1.2 Plan

The testing of the module will be carried out by a test bench written in VHDL, working in cooperation with a Matlab script. The module under test will be simulated by ModelSim, using a buffer implemented as internal block RAM on a Cyclone III FPGA, which limits the possible size of the module. A test image of 128x128 pixels will therefore be used.

The test bench will use the textio package to do reading from and writing to text files. In one process, the test bench will read input values from one file, which is converted to *std_logic_vector* and applied to the input of the module under test. A different process reads the output from the module, and writes this to a different text file when datavalid is high.

The test bench itself does no testing to ensure that the data generated is correct, but the file with output values can be read in Matlab and converted back to a .png image (for instance). In Matlab, the actual module output can be compared with the expected output from the Matlab model. Because this model is not bit-accurate, we cannot expect it to match the result of the hardware unit exactly, but the differences should nevertheless be small.

The Matlab script mentioned earlier is responsible for the reading of the output text file, as well as writing the input file. It also makes a difference image between the actual output and the model (expected) output.

As stated above, the test focuses on the main concepts of the algorithm. There are many special conditions that are not tested thoroughly. This includes behaviour when the module execution is stopped by ready, the use of the signals to define input resolution (except when they are set to maximum allowed size), and the transition from one frame to the next. The latter point depends largely on the buffer, and would therefore need to be investigated again if the buffer solution is changed.

7.2 Test Results

Before starting the test, correct perspective factors needs to be calculated for this image size. This is done according to formula 4.

This gives the following parameters:

$$
\begin{aligned}
P_x &= \frac{1.672}{128^2} && = 0.000102 \\
P_y &= \frac{0.608}{128} && = 0.00475 \\
Py_slope_start &= \frac{P_y(128-1)}{2} && = 0.3016
\end{aligned}
\tag{5}
$$

However, after trying these parameters, it was found that the image was scaled too little in the horizontal direction, suggesting that P_x should be larger. After experimenting, a value of 0.0089 was found to be more suitable for P_x. This means that the following input was applied to the module:

```
Px             <= "00010010";
Py             <= "00010010";
Py_slope_start <= "00010011";
```

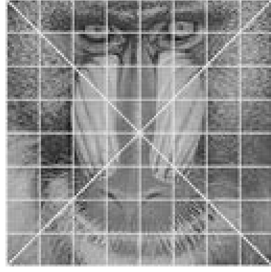
With these variables, the image in figure 15d was created from figure 15a.

By comparing figure 15d and 15c, and especially the difference between them (figure 15b) we can assess the success of the scaler. The two images are indeed quite similar, and as similar as expected given that the error concerning the formula for P_x means that the horizontal scaling in the two cases is not identical.

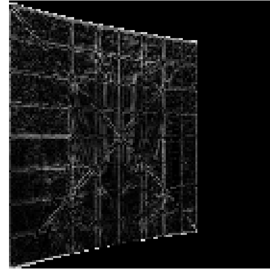
By close inspection of the right part of figure 15d, a column can be seen of non-black pixels. The cause for this should be investigated, but unfortunately time does not allow to do it within this report.

7.3 Synthesis

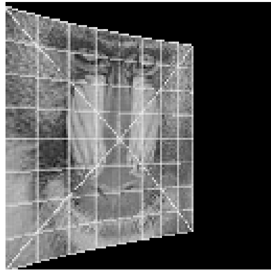
The module have been synthesised for a Cyclone III FPGA using Quartus II. The module has been implemented with an internal RAM suitable for a 128x128 pixels picture, which is the same as has been used for the simulation. The other aspects of the module, however, are scaled up to accommodate resolutions up to full HD. This does however mean that some additional logic will need to be used for the RAM controller of a bigger implementation.



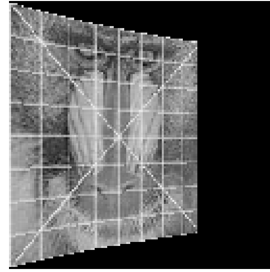
(a) Original hardware test image



(b) Difference between hardware module output and model output.



(c) Expected output generated from the model.



(d) Actual output given by the hardware module.

This discussion will focus on three design attributes: maximum frequency, total logic elements, and total memory bits.

Table 2: Overview over the synthesis results after testing the perspective scaler module with three different image sizes.

Design	128x128	256x256	512x512	1024x1024	2048x1024	2048x2048
Logic Elements	198	209	222	231	237	246
Max Frequency	287 MHz	272 MHz	265 MHz	246 MHz	243 MHz	262 MHz

Memory Bits. The module uses memory for the frame buffer, and the number of bits required is therefore easily calculated by the following equation:

$$\text{memory bits} = \text{image width} \cdot \text{image height} \cdot \text{pixel depth} = 256 \cdot 256 \cdot 8 = 524288 \quad (6)$$

This means, of course, that the memory usage increases with pixel dimensions squared, and it is evident that a frame buffer large enough for any decently sized scaler would be too big to fit on the on-chip RAM.

This means that an external DDR block, or similar storage unit, would be required. This is already used for the polyphase scaler, and using the same frame buffer has also been discussed previously. With this in mind, we conclude that memory usage will not be a major concern for the feasibility of this design.

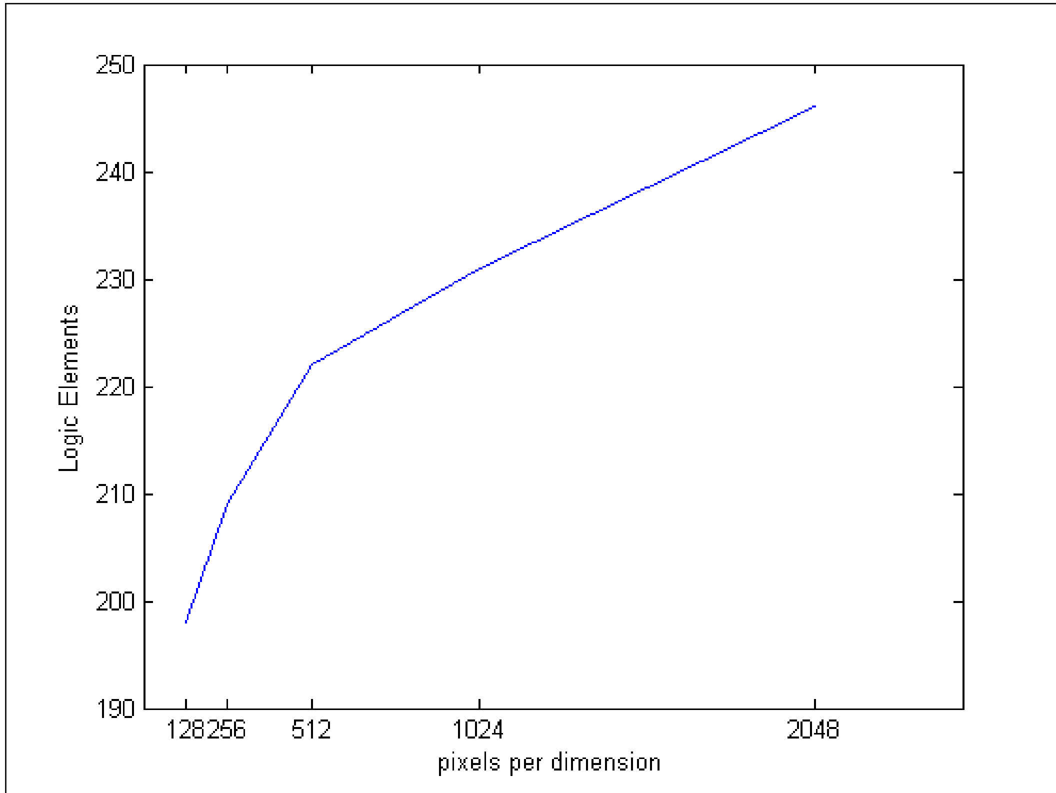


Figure 16: The number of logic elements plotted vs the scalers resolution

Logic Elements. The number of logic elements is the most important attribute to determine the complexity of an FPGA design. A simple model for how this depends on input size of generic inputs is to assume the module is built up by two parts: one part that is independent of input size, and one part that depends on it. For small input size, the constant component will dominate, while for larger input, the size-dependent component will dominate.

The logic in this design is largely found in the mapper. This analysis will therefore focus on that module. The mapper is dominated by additive operations on registers of different sizes (see figure 13). Some registers do not depend on input size (such as P_x), and some do (i.e. $xPos$). It is therefore likely that these parts will constitute the independent and the size-dependent parts of the design respectively. The transistor count in an adder is known to increase linearly with data size. As the size of the registers are not growing too fast, the growth of the design shouldn't be too steep. figure 16 confirms this with actual synthesis data.

The largest circuit configuration takes 246 LE's, which is merely a few percentages of the total FPGA. This means that the selected algorithm is efficient in terms of implementation area, and even with some logic for buffer control, it would be a quite small design.

Maximum Frequency. The clock speed of the module is estimated by Quartus using two different models; the "Slow 0 ℃" and the "Slow 85 ℃." These models represent two

different worst case scenarios, and for this report, the slowest one, the 85 °C-model, is used.

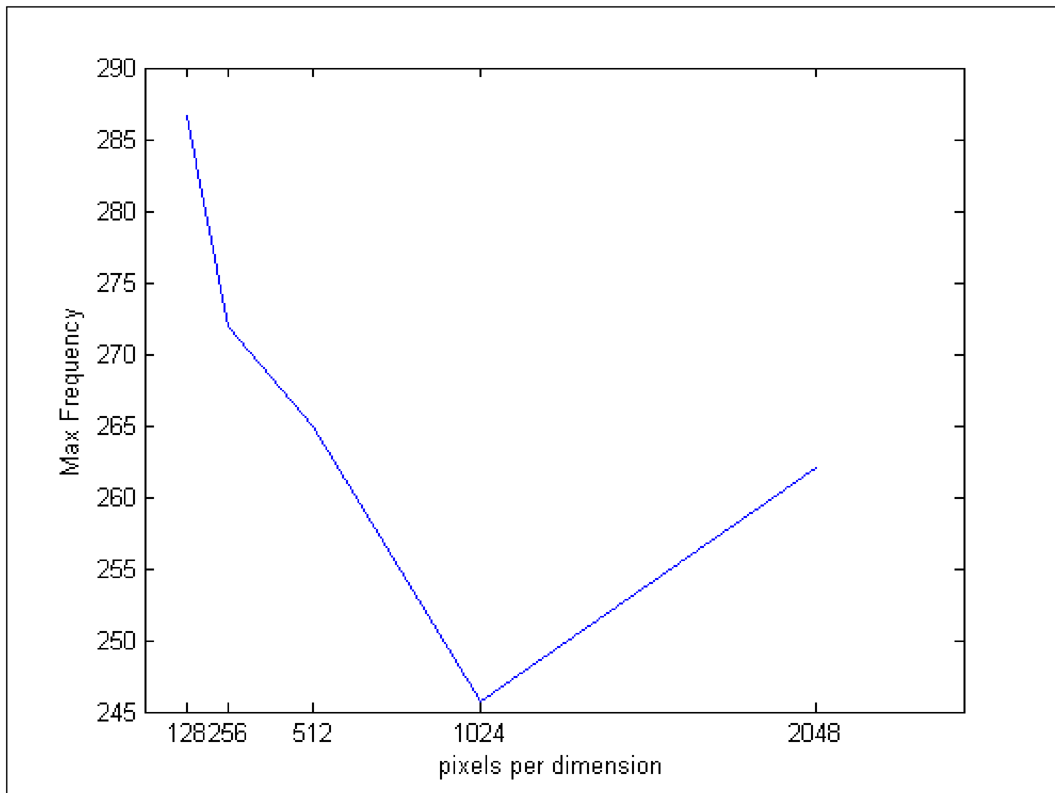


Figure 17: The maximum clock frequency plotted vs the scalers resolution

From the data in table 2, which is also represented in figure 17, we see that the circuit generally gets slower for larger implementations, which is to be expected. The slowest circuit seems to be the 1024x1024 pixel version, however, and not the largest one with 2024x2024 pixels, as would have been expected. No explanation for this result has been found, but it might be caused by different optimization choices done by the synthesis tool.

In any case, all configurations have speeds far exceeding the requirement to do real time high definition video scaling.

7.4 Specifications and Summary

The problem description specifies that the perspective scaler should operate in real time on high resolution input. In order to operate in real time, it is necessary to have an average throughput of one pixel pr clock cycle. Additionally the clock frequency must be higher than 148.5 MHz. Further, the problem description emphasises area and visual quality as important quality metrics.

The current module does not fulfil the real time requirement. This is because it only has one frame buffer, and therefore needs to first write the buffer, and then read and

operate. But, as discussed in section 6, this buffer is only used as a prototype. A real buffer would need to be implemented in a way that is able to supply the scaler with data while storing incoming data at the same time. This means either using the buffer used in the polyphase scaler, or making a similar design. Save for this issue with buffering, the circuit fulfils the requirement of calculating one pixel per clock cycle.

Additionally, the clock frequency of the module far exceeds the demand of 148.5 MHz. These two requirements ensures that the module fast enough to operate on real time video, given that a proper buffer is designed.

The module is implemented with the simplest possible algorithms, and as such, the module does not give supreme visual quality. Yet, the test images show that the perspective effects are easily recognized, and the simple filtering does not distort the image too much. Additionally, if higher quality is desired, there are alternative algorithms that could be implemented at a reasonable area cost. The current area cost is very small, and adding some more complexity should therefore not cause many problems.

The bug that is causing the two right-most lines to have wrong pixel values, however, is impacting the visual quality. The cause for this will need to be investigated before the module can be put into use.

Although some errors have been found, the testing has shown that the important concepts of the design are working. Unfortunately, time did not allow to correct these errors during the scope of the thesis, so they will need to be addressed if the scaler is to be used in a complete design.

The problem description emphasised that the design should be implemented on FPGA. This has not been carried out, as it requires much time to be spent on design of surrounding system which is not relevant for the sake of perspective scaling. It was therefore concluded that a more concentrated focus on the scaler was more in the interest of the thesis. The other requirements are considered to be fulfilled.

The biggest weakness of the design would be the visual quality. This comes as no surprise, given that the simplest algorithms are chosen for the implementation. By using more complex algorithms for mapping and interpolation, this is expected to improve considerably. Nevertheless, the relation between visual quality and implementation size is thought to be quite good.

8 Conclusion

This thesis has developed a perspective mapping algorithm called the iterative algorithm, and compared it to a state of the art geometrical image transformation algorithm, called the matrix algorithm. Additionally, three interpolation algorithms have been introduced and compared. All algorithms are modelled in Matlab.

A perspective video scaler has been implemented in VHDL based on a selection of the algorithms mentioned above. It is simulated using ModelSim, and synthesised using Quartus II.

By comparing the Matlab model of the iterative algorithm with that of the matrix algorithm, some deviation is found between their operations. Some of the causes of these deviations have been found, and suggestions for a fix is discussed. But for other parts, the exact reason is still not known. The overall visual impression of the iterative algorithm is nonetheless good, and the deviations, while visible, are not too damaging.

The unit implemented in VHDL largely conforms with the Matlab model, but two minor errors have been found. The first error relates only to the variables that are controlling the perspective operation. The formulas given to calculate these values are incorrect. This might make it harder to find the correct variables. It does not, however, affect the scaler once the correct variables are found. The second error relates to the last two columns of the output frame of the scaler. Despite of these errors, the simulation shows that the design algorithm is working, and that it is able to fulfil the requirements set forth in the problem description.

The synthesis shows that the scaler will take up little FPGA area, and is sufficiently fast, even for high video resolutions.

8.1 Future Work

Depending on the desired use of the perspective scaler, the thesis may be expanded in several ways.

The perspective scaler designed in this thesis may be expanded by substituting one or more of the modules therein. By using the perspective scaler as a starting point, a low-resource perspective scaler can be achieved with an output that is a close approximation to the mathematically correct mapping achieved by the matrix algorithm.

If it is desired to have more control over the applied transformation, the matrix algorithm could be implemented. This would give a scaler that is able to carry out a wide range of geometrical transforms in addition to perspective scaling. The downside with this is that the module would require much larger hardware resources.

If neither of these solutions are acceptable, a new algorithm will need to be developed. By improving the horizontal mapping of the iterative algorithm, and exploring the mathematical relation between matrix-indices and perspective factor variables, it is possible to bridge the gap between the two algorithms. Another alternative is to use the matrix algorithm as a starting point, and try to simplify the calculations required here. These

are only speculations, however, and the design costs and feasibility of these approaches are difficult to estimate.

A Appendix

Enclosed with this thesis are the following digital copies:

- The Matlab code defining the software model.
- The VHDL code defining the hardware design
- The test images used and the output generated.

References

[1] Svein E. Lindø, *Efficient Video Scaling Algorithms Implemented and Optimized for FPGA*. NTNU, 2011.

[2] Mark J. Burge Manson Burger, *Principles of Digitl Image Processing: Core Algorithms*. Springer, 2010.

[3] Eivind Karlsen *3D Perspective Video Scaling Effects* IET, NTNU, 2012

[4] Altera *Avalon Interface Specifications* `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=avalon%20streaming%20specification`, 2013

[5] Donald G. Bailey *Design for Embedded Image Processing on FPGAs* John Wiley & Sons, 2011

List of Figures

1 Illustration of the different phases of a 1D 4-phase polyphase filter with 4 taps, with a row of pixels illustrated below. The filter has four sets of coefficient functions (the dotted curves). For each interpolation point, one of these are selected. In the example, an interpolation point is given (red circle), and the appropriate phase is selected (the red curve). The black circles shows what values of the function that needs to be stored, and that are multiplied with the pixel values 8
2 A visualization of a picture, viewed from a camera at the top, rotated by an angle θ compared to the original position. The original position is marked by the red frame, and the view rect of the camera is given by the white lines outside of the red square. Figure form [3]. 9
3 The separation of the filter into a horizontal and a vertical part, and the buffers required in each case. The dashed circle shows the pixels that are used by the filter for one operation (the filtering window). 12
4 Shows the five 4-by-1 filtering operations that corresponds to one 4-by-4 filtering. The red pixels are scaled horizontally, and each of them ”contains” the result of an interpolation of four pixels (symbolized with the arrows). When they are scaled by the vertical scaler, the resulting operation is equivalent with a 4-by-4 interpolation with coefficients equal to the multiplication of the x- and y-coefficient of the 1-D interpolation. Note that the 1-D operations are not carried out in this order. 13
5 Simplified system architecture for the polyphase scaler. 13
6 Architecture for perspective mapping calculations in the horizontal dimension 17
7 Architecture for the perspective mapping calculations in the vertical direction 18
8 The matrix mapping algorithm that is used as a gold standard for the visual comparison. a) shows a test image scaled with the matrix mapping algorithm and the bilinear interpolation algorithm, and b) shows a visualization of the mapping of the matrix algorithm. The black wire-frame represents a 12x12 pixel input image, and the circles represent the pixel positions of the output image as given by the mapper (the interpolation points, in other words). The circles that are outside the borders of the input image makes out the black part of the image. 20
9 The simple iterated algorithm, without multiplications. a) shows a test image scaled with the algorithm along with a bilinear interpolation filter, b) shows the mapping of the algorithm (blue) compared to the matrix algorithm (red). Notice how the rows in the mapping is bending in towards the middle, resulting in a stretched look in the image. 20
10 The improved iterated algorithm. a) shows a test image scaled with the algorithm along with a bilinear interpolation filter, b) shows the mapping of the algorithm (blue) compared to the matrix algorithm (red). The error in the vertical part is corrected, but the change in the scaling factor (the distance between the circles) in the x-direction does not match the matrix algorithm completely. 21

11 A comparison between the bilinear and the nearest neighbour algorithm. Both images are scaled with the matrix mapping algorithm. 22
12 High level block diagram and interface of the hardware unit. 24
13 Overview over the range and bit length of all mapper variables. 25
14 Block diagram for the bilinear architecture 27
16 The number of logic elements plotted vs the scalers resolution 31
17 The maximum clock frequency plotted vs the scalers resolution 32