Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2006

MSc THESIS

FPGA based accelerator for real-time skin segmentation

Bart de Ruijsscher

CE-MS-2006-08

Abstract

Many real-time image processing applications are confronted with performance limitations when implemented in software. The skin segmentation algorithm utilized in the hand gesture recognition system as developed by the ICT department of Delft University of Technology presents an example of such an application. This thesis project proposes the design of an FPGA based accelerator which alleviates the host PC's computational power required for real-time skin segmentation. The communication between the host PC and the accelerator is based on direct ethernet network connection, making the proposed accelerator highly portable with a wide range of host PC's (both desktops and laptops). Moreover, the proposed framework is not limited only to the implemented skin segmentation algorithm since the same architecture can be reused for acceleration of other image processing operations. We show that our design utilizes no more than 88% of the resources available within the targeted XC2VP30 device. In addition, our solution accelerates the skin segmentation algorithm by a factor of 1.43 compared to its software implementation.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

FPGA based accelerator for real-time skin segmentation

a flexible gesture recognition system accelerator

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Bart de Ruijsscher
born in Oostburg, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

FPGA based accelerator for real-time skin segmentation

by Bart de Ruijsscher

Abstract

Many real-time image processing applications are confronted with performance limitations when implemented in software. The skin segmentation algorithm utilized in the hand gesture recognition system as developed by the ICT department of Delft University of Technology presents an example of such an application. This thesis project proposes the design of an FPGA based accelerator which alleviates the host PC's computational power required for real-time skin segmentation. The communication between the host PC and the accelerator is based on direct ethernet network connection, making the proposed accelerator highly portable with a wide range of host PC's (both desktops and laptops). Moreover, the proposed framework is not limited only to the implemented skin segmentation algorithm since the same architecture can be reused for acceleration of other image processing operations. We show that our design utilizes no more than 88% of the resources available within the targeted XC2VP30 device. In addition, our solution accelerates the skin segmentation algorithm by a factor of 1.43 compared to its software implementation.

Laboratory : Computer Engineering
Codenumber : CE-MS-2006-08

Committee Members :

Advisor:	Georgi Gaydadjiev, CE, TU Delft
Advisor:	Jeroen Lichtenauer, ICT, TU Delft
Chairperson:	Stamatis Vassiliadis, CE, TU Delft
Member:	Emile Hendriks, ICT, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

It was a great pleasure performing my thesis project at both the Computer Engineering laboratory and the ICT Group of Delft University of Technology. I am grateful that Emile Hendriks initially provided me with this challenging assignment. I would like to thank my advisors Georgi Gaydadjiev and Jeroen Lichtenauer for their professional assistance and guidance throughout this project. Additionally, they spent long hours during their spare time critically reviewing my paper and report. I really appreciate that.

Furthermore, the support of our Computer Engineering "in-house" system administrator Bert Meijs was much appreciated. All the software reinstalls, updates and additions did not seem to tire him at all.

I would also like to extend my gratitude to my family for providing all the preconditions necessary to complete my studies. Without them, it would not have been possible at all. In particular, I would like to thank Manon for providing unconditional support and interest in my work.

Bart de Ruijsscher
Delft, The Netherlands
June 19, 2006

Introduction 1

1.1 Hand gesture recognition

Human-computer interaction (HCI) is a discipline involved with the design, implementation and evaluation of interactive computer systems which provide the user with natural and efficient means for interaction. One of the known human communication modes is using hand gestures. Systems capable of recognizing gestures are envisioned to provide more natural user interaction compared to systems relying only on keyboard and/or mouse user input. Application domains for hand gesture recognition include input and control of user applications and games, interactive training of sign languages and behavior monitoring.

1.2 Problem statement

The Mediamatics department, part of the Electrical Engineering Mathematics and Computer Science faculty of Delft University of Technology, is currently developing a gesture recognition system based on images captured by a digital video camera. The current system is implemented in software and is unable to achieve the frame processing rate of 25 frames per second as is required for accurate hand gesture recognition. The primary cause for this can be attributed to the *skin segmentation* algorithm computational overhead employed in the gesture recognition system. Currently, the system works with an image resolution of 160x120 pixels at a frame rate of 25 images per second. It is desirable to improve the image resolution up to 640x480 pixels at the same frame rate. The higher resolution is desirable since it will make it possible to determine the shape of a human hand, to perform better tracking of features and to have more freedom of movement for the user. This allows larger variations in distance between the user and the camera. Furthermore, a higher resolution would allow the use of better methods of gesture recognition since more details (such as seperate fingers) will be available.

Since the skin segmentation algorithm [22] is still under development, the definitive version has not yet been established. Consequently, in the future new operations might be included and some currently implemented operations might be excluded in the following versions of the algorithm.

The main problem statement which presents the basic motivation for this thesis project can therefore be formulated as follows:

How can the implementation of the existing skin segmentation algorithm be accelerated such that the gesture recognition system is able to operate at a video frame rate of 25 frames per second and a resolution of $640x480$ *pixels? Furthermore, how can this implementation be achieved while providing the flexibility to support new versions of the algorithm in the future?*

1.3 Objective and plan of approach

The objective of this thesis project is to accelerate the skin segmentation algorithm in order to meet the real-time requirements of the gesture recognition system at the desired image resolution and frame rate. A proof of concept has to be developed to demonstrate the operation of the proposed system. Figure 1.1 shows the conceptual representation of this thesis project. The PC captures images from a digital camera and sends these to the accelerator. The accelerator processes the images as required by the skin segmentation algorithm [22] and returns both a skin segmented image and an absolute difference image. High level applications can use the accelerator output for further processing. For example, a hand gesture recognition application may analyze the accelerator output to interpret hand gestures. The interpretation may result in application specific feedback such as character movement in a game or sign matching in a sign language training application.

A contextual black-box representation of the accelerator is depicted in figure 1.2. The figure shows that a stream of RGB images constitutes the input to the accelerator. The output consists of both an absolute difference image stream and a skin segmented image stream. The absolute difference image is used for change detection between subsequent frames, the skin segmented output is used for hand gesture recognition.

The plan of approach consists of the following phases:

- Literature study, both on hand gesture recognition and real-time image processing related systems;
- Selection of a suitable implementation platform;
- Architectural exploration and development of suitable processing architecture;
- Implementation, testing and performance evaluation of the architecture and its components;
- Development of a demonstrator.

1.4 Chapter overview

The remainder of this document is organized as follows. Chapter 2 presents an overview of previously reported work related to this project. Chapter 3 provides a description of the image processing operations currently used in the skin segmentation algorithm. Chapter 4 discusses aspects important to the implementation of the accelerator and concludes with a motivated choice for a suitable implementation platform. Chapter 5 discusses the accelerator architecture and chapter 6 provides a detailed discussion on the implementation of this architecture and its components. Chapter 7 provides the system evaluation, chapter 8 then concludes with a summary of this project.

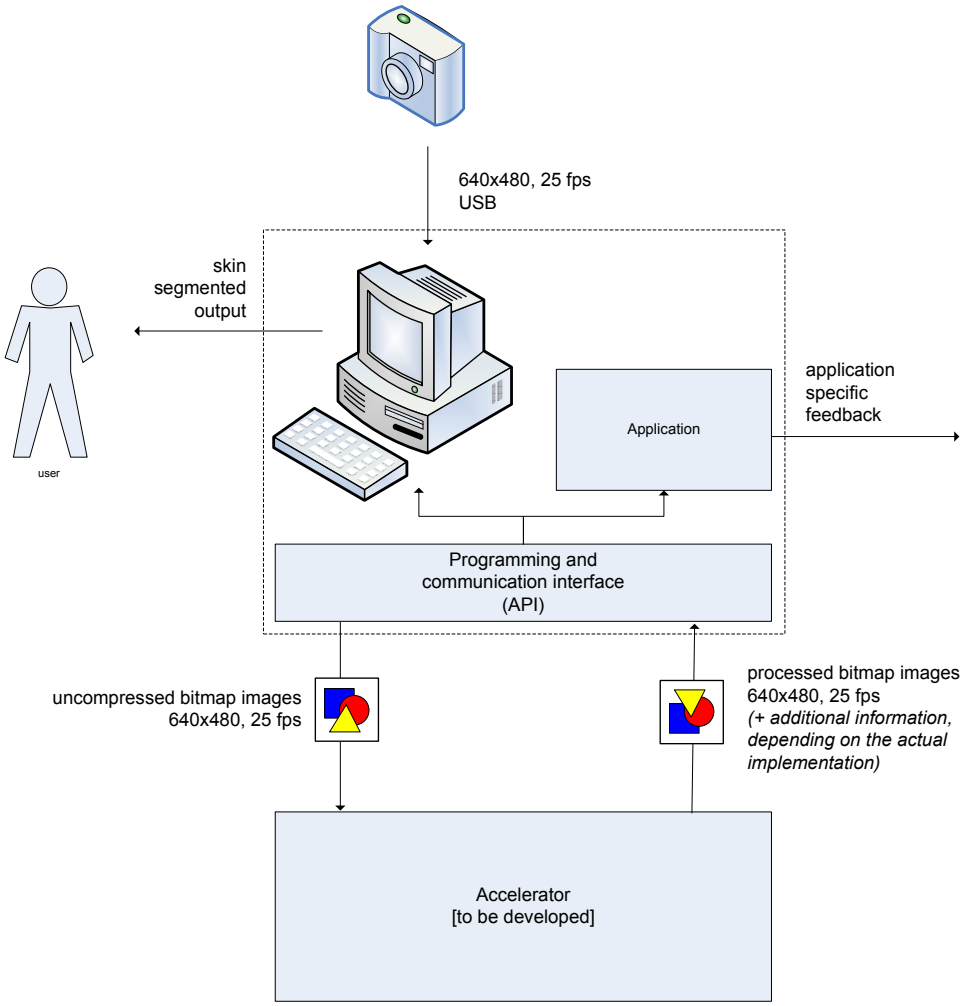

Figure 1.1: project overview

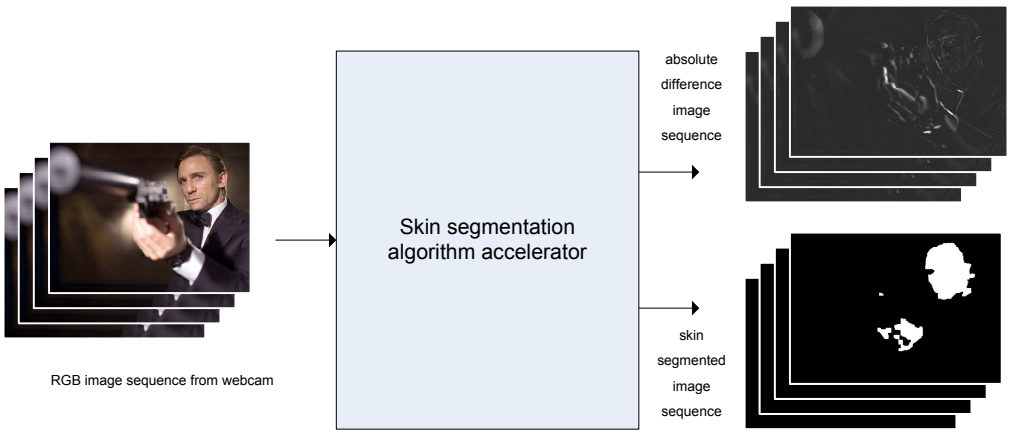

Figure 1.2: project black box representation

Related work 2

This chapter discusses previous related work. First, a short overview of real-time skin segmentation methods is discussed. Then, an overview of known Field Programmable Gate Array (FPGA) based solutions for accelerating real-time image processing operations is presented.

2.1 Real-time skin segmentation

Many computer vision based methods for hand gesture recognition or skin color tracking have previously been proposed in the literature. An interesting method for tracking multiple skin colored objects proposed in [2], shows that it is possible to achieve real-time performance of the proposed algorithm. This skin segmentation algorithm is based on transforming the 3D color representation (YUV) of input images to a 2D representation (UV) and calculating the probability of each pixel being a skin color. The reference for this calculation is determined by a set of training images which has to be constructed manually. The skin regions of the image are then grown into blobs[1] which can be tracked through time. The disadvantage is the requirement of manual composition of the training set.

The discussion of a different approach proposed in [18] also claims to provide real-time performance of the skin segmentation algorithm. This work however lacks information on the computational complexity of the algorithm, the video resolution and the frame rate utilized. The method uses a look-up table of skin colors based on a predefined training set. During run-time the skin segmentation is performed on a low resolution input image. One disadvantage of this method is the low resolution input image used for skin segmentation. Such reduced resolution prevents detection of details such as fingers and hand tilt.

To reduce the computational complexity of hand gesture recognition, several researchers have proposed alternative methods for hand tracking. For example, [19] proposes a method based on "flocks of features" to track skin colored objects. The method calculates feature points within the hand area of an image and tracks these points over time. The advantage of calculating a limited amount of feature points is the reduction of the computational complexity compared to calculations involving the complete input image. The disadvantage is the inability to precisely determine hand shape, tilt and finger positions.

A different approach as mentioned in [1] is based on histogram segmentation. The method relies on dividing the image into small regions. For each region a histogram is calculated and matched with a training set. The advantage of this algorithm is a

[1] Blobs are solid circle or elliptical shaped object which represent a simplification of the object considered.

reduction of computational complexity but this particular algorithm lacks the ability to track multiple objects in a single image.

The method utilized in this project is based on the algorithm as proposed in [22] which is currently under development at the ICT department of Delft University of Technology. It provides better results compared to other models [22] without requiring color calibration of the camera.

2.2 FPGA based acceleration of real-time image processing operations

Traditionally, application specific IC's (ASICs) have been employed to accelerate the execution of image processing operations. The main drawback of such systems is the fact that the structure of the system cannot be changed after fabrication. However, the recent developments of reconfigurable hardware provide means for acceleration of applications without compromising on flexibility.

Several PCI based implementations of reconfigurable hardware coprocessors for real-time image processing have been proposed. For example, [28] proposes an architecture based on specialized image processing modules which can be configured in the FPGA. This method however only enables a few modules to be used at a certain moment, which is a limitation in case of complex algorithms like the skin segmentation algorithm targeted in this project.

A different approach is proposed in [27] and presents a coprocessor specifically designed to accelerate the addressing of pixels within software applications. The disadvantage of this approach is the limited resolution and means to implement complex image processing operations.

Several authors have proposed a high-level approach for the problem of executing image processing operations in real-time. For example, [24] proposes a method of describing image processing applications using single assignment C (SA-C). Such a description can be converted to a dataflow graph by a specialized compiler and implemented in reconfigurable hardware. Another high level method was proposed in [7] and essentially comprises a specialized compiler based on a set of predefined FPGA configurations.

Customized architectures designed for specific applications have been also discussed in the literature. For example, an architecture for computer vision based navigation is proposed in [4]. It accelerates the calculation of image processing operations for navigation by utilizing several FPGA boards simultaneously.

An FPGA implementation of a pixel processor for object detection applications has been discussed in [23]. The proposed architecture accelerates the implementation of object detection by implementing the computationally intensive parts of only one specific algorithm in hardware.

The implementation of 2-D feature detection on an array of FPGAs has been presented in [3]. The authors present their implementation of a commonly utilized algorithm for feature detection on an array of FPGAs. The system is able to track features of input images in real-time at an image resolution of 320x240 pixels.

2.3 Conclusion

The discussion in this chapter shows that previous work related to reconfigurable hardware acceleration of image processing operations has been proven successful. The main disadvantage of the considered proposals is the PCI interface being employed. This limits the practical application of such accelerators to desktop systems only.

The high level approaches for describing image processing operations for reconfigurable hardware may provide interesting alternatives to low level approaches. At this moment however, the methods lack the possibility to implement complex algorithms consisting of multiple operations.

Taking into account the above mentioned aspects, we can summarize the requirements for our accelerator for real-time skin segmentation as follows:

1. Acceleration of the skin segmentation algorithm using a commonly available PC interface, available on both desktop and laptop systems;

2. Flexible architecture of the accelerator which allows the implementation of future versions of the skin segmentation algorithms;

3. Possibility to incorporate future developments in high level descriptions of image processing algorithms in the architecture.

Image processing operations 3

In order to develop a suitable accelerator, it is important to understand the computational aspects of the operations involved. This chapter therefore discusses the image processing operations utilized in the selected version of the skin segmentation algorithm.

3.1 Introduction

Images can be described as two dimensional arrays of pixels, each pixel representing a measured sample of the "real-word" brightness and color information at the corresponding coordinates. Image processing involves meaningful transformations of source images. A wide variety of different image processing operations exist and have all been thoroughly described in the literature.

Hand gesture recognition systems and - more specifically - tracking systems, involve tracking of features (blobs, active contours or articulated models). Determining these features requires spatial information. This implies utilization of image processing algorithms operating at spatial representations instead of frequency based representations.

An important method to classify the image processing operations for still images is by examining the spatial dependencies of the respective transformations:

- *point operations*: calculation of the new value depends only on the corresponding value in the source image (no neighborhood);
- *local operations*: calculation of the new value depends on the corresponding values in the (spatial) neighborhood of the source image;
- *global operations*: calculation of the new value depends on all the values in the source image.

The following sections discuss the convolution, color space conversion, erosion and dilation, local count and absolute difference operations used in the skin segmentation algorithm. Additionally, to provide a perspective on the implementation of these operations, their algorithmic representations are presented.

These can be classified either as a point operation or a local operation. The color space conversion and absolute difference calculation can be classified as point operations. The convolution process used for smoothing filters and the morphological operations such as erosion and dilation can be classified as local operations.

3.2 Convolution

This section discusses the convolution operation and two examples of smoothing filters based on the convolution operation. Furthermore, an algorithmic representation of the

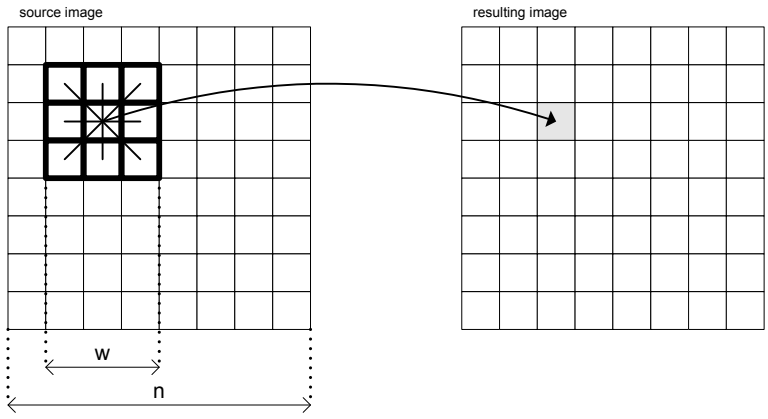

Figure 3.1: image convolution process

convolution is presented.

Convolution is an operation commonly utilized in many image processing tasks. In its mathematical definition, the convolution of two functions f and g produces a resulting function h and is denoted as follows:

$$h = f * g \tag{3.1}$$

In two dimensional continuous space, the convolution is formally defined as:

$$h(x, y) = f(x, y) * g(x, y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(q, r) g(x - q, y - r) dq dr \tag{3.2}$$

In two dimensional discrete space, which is the case in digital image processing tasks, the convolution is defined as:

$$h[x, y] = f[x, y] * g[x, y] = \sum_{q=-\infty}^{+\infty} \sum_{r=-\infty}^{+\infty} f[q, r] b[x - q, y - r] \tag{3.3}$$

From an algorithmic perspective, convolution is a local operation which scans a window across the image to calculate the output pixel values. This window is often denoted the convolution *kernel* or mask. Figure 3.1 depicts the scanning process, the kernel with a width of 3 is represented with thick black lines. If both, the width and height of a source image consist of n pixels, the time complexity of a convolution based algorithm is $O(n^2)$. The computational complexity per pixel is $O(w^2)$ with w representing the width of the kernel. Whenever the convolution filter function is separable[1], it is possible to perform two subsequent one dimensional scans instead of one two dimensional scan.

Convolution constitutes the basis for many digital image processing filter implementations, smoothing filters are commonly implemented using the convolution method. Two examples of such smoothing filters, the mean filter and the gaussian smoothing filter, are discussed in the following subsections.

[1]A function is denoted "separable" if the two independent variables can be separated

1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25

Figure 3.2: 5x5 mean filter kernel centered around (0,0)

3.2.1 Mean filter

A mean filter calculates a new pixel value by averaging over the neighborhood in the source image utilizing a smoothing kernel with equal kernel weights. For example, smoothing using a 5x5 mean filter kernel can be accomplished using equation 3.4.

$$S(x,y) = \frac{\sum_{j=-2}^{2}\sum_{i=-2}^{2} w_{(i,j)} * p_{(x+i,y+i)}}{25} \tag{3.4}$$

where S denotes the smoothed pixel, w denotes the kernel coefficient at position (i,j) and p denotes the pixel at neighborhood position $(x+i, x+j)$. The kernel of a 5x5 mean filter is depicted in figure 3.2. As can be seen in this figure, the kernel coefficients have equal weights. In the case of a 5x5 kernel, the weights are $1/(5*5) = 1/25$. The resulting pixel value contains the mean of the pixels in the source image's neighborhood.

3.2.2 Gaussian smoothing filter

The Gaussian smoothing filter is a second example of a smoothing filter with kernel weights based on the Gaussian distribution function:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{3.5}$$

The gaussian function G in equation 3.5 at x also depends on the parameter σ which determines the sharpness of this function.

For image processing operations, a two dimensional isotropic Gaussian function is employed:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.6}$$

where G is the gaussian kernel weight at the location with coordinates x and y. The σ parameter again determines the sharpness of the gaussian function. This function is quantized into discrete values in order to develop a convolution kernel of a specific size.

0,002969	0,013306	0,021938	0,013306	0,002969
0,013306	0,059634	0,09832	0,059634	0,013306
0,021938	0,09832	0,1621	0,09832	0,021938
0,013306	0,059634	0,09832	0,059634	0,013306
0,002969	0,013306	0,021938	0,013306	0,002969

Figure 3.3: 5x5 gaussian filter kernel with $\sigma = 2$ centered around (0,0)

Figure 3.4: application of mean filter (middle) and gaussian filter (right)

For example, a 5x5 Gaussian kernel with $\sigma = 2$ can be represented as depicted in figure 3.3. The kernel weights shown in the figure have been calculated using equation 3.6.

Application of both mean filter and gaussian filter to an example image yields the result as shown in figure 3.4. The left image shows the original image, the middle image shows the application of the mean filter and the right image shows the application of the gaussian filter. The right image shows a little more detail compared to the middle image but requires more complex calculations.

3.2.3 Algorithmic representation of the convolution operation

Listing 3.1 provides the algorithmic representation of the convolution operation. Using this algorithm, the source image s is scanned on a row first basis. The output pixel value in d is calculated by multiplying the corresponding neighborhood pixels in s with the kernel weights in w. An odd-sized square kernel centered around $(0, 0)$ is assumed, hence it is indexed (both horizontally and vertically) from $-(w.width - 1)/2$ to $+(w.width - 1)/2)$.

Listing 3.1: Pseudo code for convolution operation

```
%% s = source image
%% d = destination image
%% w = kernel window

for (i,j) := 1 to (s.height, s.width) do
begin
    for (m,n) := (-(w.height-1)/2, -(w.width-1)/2) to
                 +(w.height-1)/2, +(w.width-1)/2) do
    begin
        d[i,j] = s[i+m,j+n] * w[m,n];
    end;
end;
```

3.3 Color space conversion

A color space essentially encompasses all possible combinations of the color components of a given model, which can be any vector *representation* of color. The red, green and blue (RGB) color space presents an example of a regularly utilized color space. Each color within this color space can be represented by a single RGB vector.

Color space conversion involves the process of converting a given color representation into another representation by a mapping function. An example of such a mapping is the RGB to Cyan Magenta Yellow (CMY) color space conversion is shown in 3.7.

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{3.7}$$

A conversion employed in the current skin segmentation algorithm converts the RGB representation into a greyscale representation using the average of the R,G and B components as follows:

$$I(x) = \begin{bmatrix} R(x) & G(x) & B(x) \end{bmatrix} \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \tag{3.8}$$

As can be seen in equation 3.8, the RGB to grayscale conversion process converts a 3-component input value to a single component output value. An algorithm representation of this conversion process is given in listing 3.2.

Listing 3.2: Pseudo code for RGB to greyscale color space conversion operation

```
%% s = source image
%% d = destination image

for (i,j) := 1 to (s.height, s.width) do
```

```
begin
    d(i,j) := (1/3) * s(i,j).R + (1/3) * s(i,j).G + (1/3) * s(i,j).B
end;
```

3.4 Erosion and dilation

Erosion and *dilation* are two examples of morphological operations. Applying these operations changes the structure or form of an image, the element used to transform an image is called a *structuring element*. Both greyscale and binary morphological operations exist. However, only the binary versions are of interest to this project since these versions are used in the skin segmentation algorithm[2].

The set resulting from the dilation of an input image A with a structuring element B can be denoted as follows:

$$A \oplus B = \{z|[\left(\hat{B}\right)_z \cap A] \subseteq A\} \tag{3.9}$$

The dilation of A by B is the set of all displacements z of B denoted $\hat{B}$, such that $\hat{B}$ and A intersect by at least one element. Algorithmically, the process of performing a dilation with a square shaped structuring element can be described as follows:

Listing 3.3: Pseudo code for a binary dilation operation

```
%% A = source image
%% B = structuring element
%% C = resulting image

for (i,j) := 1 to (A.height, A.width) do begin
    sum := 0;
    for (m,n) := (-(B.height-1)/2, -(B.width-1)/2) to
                 +(B.height-1)/2, +(B.width-1)/2) do
    begin
        sum := sum + A[i+m, j+n];
    end;

    if (sum > 0) then
        C[i,j] := 1;
    else
        C[i,j] := 0;
    end if;
end;
```

The binary erosion of an input image A with a structuring element B can be denoted as follows:

[2]The erosion and dilation operations used in the skin segmentation algorithm operate on the output of threshold operations. Therefore, only binary erosion and dilation are of interest to this project.

$$A \ominus B = \{z | (B)_z \subseteq A\} \tag{3.10}$$

Thus, the erosion of A by B is the set of all displacements z of B such that B and A overlap by all elements. The algorithmic representation of an erosion is similar to the dilation with the exception that the if-condition should read:

Listing 3.4: Pseudo code excerpt for a binary erosion operation

```
...
    if (sum = B.size) then
        C[i,j] := 1;
    else
        C[i,j] := 0;
    end if;
...
```

The binary *opening* of an image can be achieved by first applying an erosion followed by a dilation. This operation effectively smoothes the the contour of objects in the image. When applying a dilation followed by an erosion, the result consists of the binary *closing* of that image. This operation also smoothes the contours of objects but fuses narrow breaks in the contours.

Figure 3.5 shows the result of an input image (a) processed with a dilation (b), erosion (c), opening (d) and closing (e). For a detailed discussion on mathematical morphology, please refer to [11].

3.5 Local count and absolute difference

The binary local count filter scans a kernel across the binary input image and counts the amount of pixels with a value of 1 within the neighborhood. To a certain extend, the effect of this operation is similar to that of a uniform filter with kernel weights of 1. The uniform filter however calculates the average of the pixels in the neighborhood, whereas the local count calculates the sum of these pixels.

To provide a basic method of motion estimation, the skin segmentation also employs the computation of the absolute difference between two grey scale images based on 3.11.

$$D(x) = |I_t(x) - I_{t-1}(x)| \tag{3.11}$$

where $D(x)$ denotes the absolute difference at location x, $I_t(x)$ and $I_{t-1}(x)$ denote the intensity of a pixel at location x in a frame at time t and $t-1$ respectively.

It provides information on the intensity change between two subsequent grey scale images. As opposed to the local count operation, this is a point operation and thus it does not require the neighborhood of each pixel to calculate the output pixel.

3.6 Summary

The image processing operations discussed in this chapter are used in the targeted skin segmentation algorithm. These operations can be classified either as point operations or

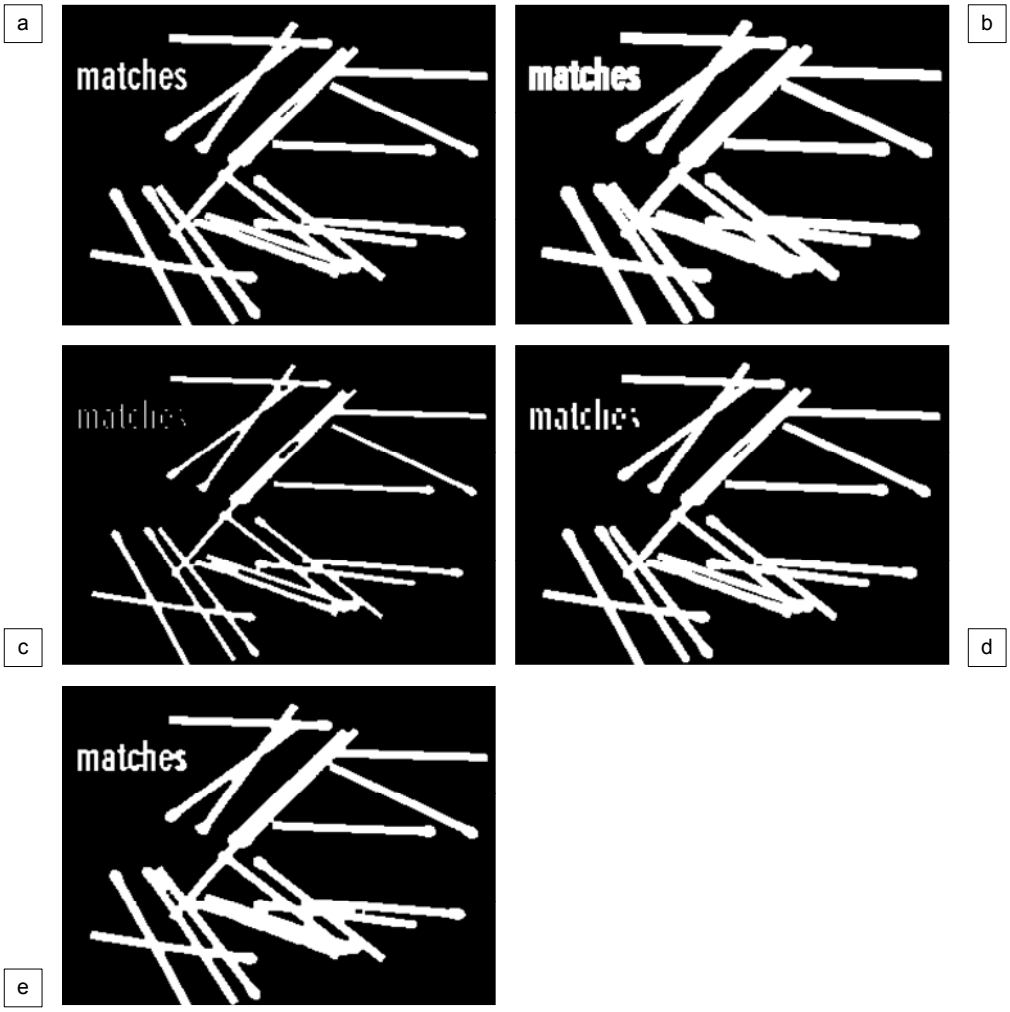

Figure 3.5: application of a dilation (b), erosion (c), opening (d) and closing (e) operation

local operations. Thus the computation of the results require either only the input pixel or its neighborhood (defined by the kernel of the operation). It is worth noting that the local operations share a similar window scanning method.

Implementation considerations 4

The mapping of image processing operations such as those mentioned in the previous chapter onto a suitable platform requires knowledge of the related implementation aspects. This chapter introduces the aspects and concepts related to parallel processing systems, flexibility and possible candidate implementation platforms. A choice for the most suitable platform is presented at the end of this chapter.

4.1 Exploiting Parallelism

Through the exploitation of inherent parallelism, operations can be executed considerably faster compared to sequential implementations. Many image processing operations are suited for implementation utilizing a certain type of parallelism because of the lack of data dependencies. Before discussing appropriate implementation platforms, a brief overview of relevant parallelism concepts is presented. First, instruction level parallelism is introduced. This is a type of parallelism supported by modern processors and thus is being used in the current software implementation of the skin segmentation algorithm. Second, task level parallelism is introduced. This is a type of parallelism as perceived from an application perspective. The discussion is by no means intended to be a comprehensive and complete overview. For a detailed discussion, the reader is referred to literature on parallel processing such as [8] and [12].

4.1.1 Instruction level parallelism

Probably the most popular taxonomy on instruction level parallelism was proposed by Flynn [9] in 1966. It introduces the following categorization based on instruction and data streams:

- single instruction stream, single data stream (SISD): this category is also denoted as 'uniprocessors' and utilize a single instruction which operates on a single data set;
- single instruction stream, multiple data stream (SIMD): a single instruction operates on multiple data sets, multimedia extensions of modern processors are often SIMD style instruction;
- multiple instruction, single data (MISD): multiple instructions operate on a single data set, this is a quite uncommon category;
- multiple instruction, multiple data (MIMD): multiple instructions operate on multiple data sets, this category comprises multiprocessor systems which utilize multiple processors.

The general purpose uniprocessor presents an example of the commonly utilized SISD category. The traditional operation of such processors consists of the fetch-decode-execute-write back or a somehow similar execution cycle. First, an instruction is fetched from memory, then it is decoded and the relevant functional unit[1] is addressed, the execution takes place in the functional unit and after completion the results are written back into memory. This execution cycle allows for a technique denoted pipelining. This technique allows the staged and concurrent operation inside the execution cycle. For example, at a certain moment instruction A can be decoded while simultaneously instruction B is being fetched from memory. Pipelining is a common technique utilized in many processors presently available.

It is possible to further parallelize execution through the provision of multiple parallel execution pipelines. This technique, denoted as super scalar execution, allows for an instruction to be dispatched to an available pipeline. Effectively, multiple instructions can be present in the different pipelines concurrently.

Although these techniques allow the execution of multiple instructions per clock cycle, the data dependencies between subsequent instructions and the non-linear execution flow introduced through conditional branches[2] pose a limit on the effectiveness of these techniques. Since a conditional branch can depend on the result of an instruction which is executed at the moment the branch is evaluated, the processor is unable to decide which consequent instruction to fetch.

Several initiatives to alleviate this problem exist, either depending on hardware or on software support. A technique commonly found in modern digital signal processors is based on an instruction format which encodes independent operations into a single instruction. This instruction format is denoted Very Long Instruction Word (VLIW). During execution, the operations present in the instruction are separately dispatched to different execution pipelines. It is vital that these operations do not share data dependencies, it is the task of the compiler to select operations which can be executed independently.

The concepts of parallelism present in modern processors have been discussed in the literature. For a detailed discussion of architectural aspects of parallelism, the reader is referred to [15].

4.1.2 Task level parallelism

Another level of parallelism can be found at the task level of an operation. By examining the operation to be performed, different tasks may be identified which can operate independently without introducing any data hazards.

Since this level of parallelism depends entirely on the type of operation, it is difficult to provide a general framework for the categorization of task level parallelism.

Task level parallelism can be implemented as thread level parallelism, where each task is assigned to an execution thread. This thread can then be executed on an available processing unit. An important aspect introduced by utilizing thread level parallelism is

[1] Examples of functional units are the Arithmetic and Logic Unit (ALU) and Floating Point Unit

[2] A conditional branch determines if the current execution flow can continue by evaluating a certain register or flag

the communication overhead involved. Different threads may need to communicate with an arbiter or other threads in order to successfully finish their respective operations.

Algorithmic representation often need to exhibit explicit constructs (for example the balanced tree construct) and communication operations to utilize task level parallelism. A detailed discussion of parallelism concepts and the design of parallel algorithms can be found in [12].

4.2 Flexibility

The term "flexibility" is a rather broad and subjective term. Generally, the design problem is often confronted with the compromise between flexibility and performance improvement. Flexible systems most likely offer less performance compared to less flexible but fully optimized systems. It is therefore necessary to examine the problem more closely to provide some concrete criteria for the system to be developed. In the context of this project we therefore define the following relevant concepts:

- flexibility of function; from a functional perspective the system should perform the image processing operations as defined by the "skin segmentation" algorithm employed (as described in detail in appendix A). However, this algorithm will be improved and therefore modified within the near future. It is therefore required that a next version of this algorithm can be executed by the system with as little effort as possible;
- flexibility of operation; several parts of the algorithm contain parameters which influence the operation they perform. These parameters should not be fixed in the system but a possibility should be provided to modify these parameters during system operation;
- flexibility of use; from the end-user perspective the system needs to be flexible in (daily) use. Therefore, an easy to operate or connect system is preferred over a system which requires complex installation. Judging by the context of this project however, this concept is of less importance although it should be considered if feasible.

4.3 Implementation platforms

A variety of different implementation platforms exist, each exhibiting specific characteristics which may or may not aid in the acceleration of image processing operations. The following sections provide a short overview of the possible implementation platforms and their effectiveness on the implementation of real-time image processing operations.

4.3.1 General purpose processor

General purpose processors (GPP's) provide an execution environment for general purpose processing tasks. The instruction set architecture is suited to different types of computations. Some architectures are extended with instructions for specific processing

tasks. The Pentium processor is an example of a general purpose processor with SIMD type extensions for multimedia processing. It exhibits instruction level parallelism techniques such as those mentioned in section 4.1.1.

General purpose processors provide little support for true task parallelism, although emulated parallelism can be achieved by utilizing several threads in an application. However, threads will not be executed truly in parallel since the instructions will be executed sequentially.

High level software development for general purpose processors requires little knowledge of the processor architecture. It is therefore suitable for flexible implementations of image processing operations.

4.3.2 Digital signal processor

Digital signal processors (DSPs) are especially suited for specific signal processing algorithms. Both processor and instruction set architecture utilize special constructs to optimize execution times for these algorithms. Furthermore, DSPs might use SIMD style instructions to implement parallel processing.

Software development for digital signal processors often requires knowledge of the processor architecture to develop an efficient implementation. It is therefore less suitable for flexible implementation of image processing operations compared to the general purpose processor.

The Philips Nexperia processor (also known as the Trimedia) is an example of a digital signal processor specifically designed for MPEG encoding and decoding. Its instruction set is based on the VLIW concept and thus it provides means for parallel execution of instructions.

The graphics processing unit (GPU) commonly found on PC add-on boards, is another example of a digital signal processor. These units most often find their implementation as accelerator for specialized 3D graphics processing operations.

4.3.3 Application specific IC

Processors as mentioned in the previous sections employ a fixed hardware architecture which is programmable in the sense that it is able to execute user algorithm instructions. In contrast, application specific integrated circuits (ASICs) contain optimized circuit implementations of the desired functionality in hardware.

Hardware description languages (HDLs) can be used to develop an ASIC. These languages are syntactically similar to imperative languages but provide means to describe parallel hardware circuits rather than a sequence of processor instructions. Consequently, a compiler for a hardware description language produces a hardware circuit description instead of low-level processor instructions.

Another fundamental difference is that hardware circuits inherently operate concurrently and thus provide the possibility to develop parallel structures. Moreover, because of this parallel nature, development of sequential operations requires special techniques such as finite state machines.

4.3.4 Programmable hardware

Several different types of programmable hardware exist, but the predominant type currently is the Field Programmable Gate Array (FPGA). The term "field programmable" can be attributed to the fact that once the FPGA is fabricated, its operation can be adapted "in the field". This is contrary to the ASIC which cannot be adapted once fabricated.

An FPGA contains a large amount of programmable logic cells, each cell typically containing several registers and logic and performs a variety of different functions according to its configuration. The cells are interconnected and thus provide the possibility to develop logic circuits of a larger scale.

Whereas an ASIC consists of fixed but highly optimized circuitry, FPGA's provide a flexible configuration through the utilization of the programmable logic cells and interconnections. Consequently, ASIC implementations often achieve higher performance compared to FPGA implementations.

Specification and validation of functionality may follow the same approach as for ASIC development, the actual implementation however differs in the fact that an ASIC has to be manufactured while an FPGA design may be adapted at any time.

4.4 Conclusion

From the implementation platforms presented, each has advantages and limitations with respect to efficient implementation of image processing operations. General purpose processors provide a considerable degree of flexibility but often do not allow a parallel implementation of image processing operations. Digital signal processors provide a more efficient and thus better performing implementation compared to general purpose processors, but still do not allow true parallel execution of the image processing operations. An ASIC implementation would provide the most efficient and best performing implementation but lacks the required flexibility to adapt the design when necessary. Furthermore, the cost and time associated with the manufacturing of an ASIC present an important reason not to consider it as an interesting implementation platform for our project.

Consequently, the FPGA platform yields the best balance between flexibility and performance. The design can be adapted relatively easily by modifying the source files and re-synthesizing the design. A parallel implementation is possible by developing concurrent hardware circuits (blocks) for each image processing operation.

Development board

To design and implement the accelerator on an FPGA, a suitable development board is required. This board should provide a communication interface capable of transferring the images to and from the FPGA.

After a careful investigation, we chose to utilize the Digilent XUP V2P board [14] for our accelerator. It is a low-cost FPGA board containing a Virtex II Pro FPGA and various communication interfaces. The manufacturer provides a fully functional communication stack for 10/100 Mbps ethernet interface, effectively reducing the effort required to obtain an operational communication link.

For the remainder of this thesis we will refer to the programmable hardware implementation using an FPGA simply as ”hardware implementation”.

5 Architecture design

This chapter describes the architectural design of the developed system. It also states the implementation constraints derived from the system real-time requirements.

5.1 Architecture

The architecture of the system describes the components involved, their function and their respective interfaces. The next paragraphs present a description of the overall system architecture and the accelerator architecture. First, the flexibility criteria imposed on the architecture are discussed, then the architectures of the overall system, the accelerator and pixel processing pipeline are presented.

5.1.1 Criteria for architecture design

In the previous chapter we defined the three flexibility criteria for the accelerator to be developed. From these general criteria we can derive requirements for the system and accelerator architecture.

The *flexibility of use* criterion depends both on the accelerator and on the driver software on the host PC. The accelerator should provide sufficient means to aid in the flexible use by the user. Connecting and disconnecting should therefore be straight forward and should not require much user interaction.

To provide *flexibility of function*, it should be possible to adapt the architecture of the algorithm in hardware. There are several different methods to implement such a functionality:

1. A very flexible method can be found by implementing an architecture similar to those utilized by (digital signal) processors. This will provide the highest flexibility but the least performance improvement;

2. Using a micro programmable architecture, operations can be implemented by describing each operation as a sequence of micro operations. The implementation would involve implementing the micro operations and programming the algorithm by specifying the required micro operations. This will provide a lower degree of flexibility but a higher improvement in performance compared to the general purpose architecture approach.

3. A fixed implementation where each algorithm operation is implemented directly in the reconfigurable hardware. This will provide the highest performance but the least flexibility.

We decided to implement the operations in hardware (option 3 above) since this provides the highest performance improvement. Furthermore, since we utilize an FPGA, we can implement different algorithms by redesigning only the required parts. This method is therefore not completely inflexible and still allows algorithm modifications.

Flexibility of operation presents an important criterion for the architecture. It is desirable to control the parameters of several predefined algorithm operations when the accelerator is in use. The architecture should therefore provide the possibility to access and modify such parameters.

5.1.2 System architecture

The overall system architecture is depicted in figure 1.1. The system consists of a host PC connected to the accelerator through an Ethernet network connection. The PC captures images from a webcam at a frame rate of 25 frames per second and transfers these images to the accelerator. The accelerator processes the images and returns the results back. This architecture allows *flexibility of use* since it can be connected using a regular network cable. The FPGA allows *flexibility of function* since the hardware description of the algorithm can be adapted. *Flexibility of operation* will be achieved by allowing algorithm parameters to be changed at run-time.

Considering the Ethernet network connection used and the fact that very high bandwidth utilization is essential for the overall system performance, we adopted UDP for our network communication protocol. There are two packet types: data (also referred to as *pixel packets*) that carry consecutive pixel- or accelerator output information and *configuration packets* used for system configuration at run-time.

The pixel packets are 1358 bytes long and contain 320 pixels using 4 bytes per pixel (320*4 = 1280 pixel bytes). The remaining 78 bytes are used for the following headers: accelerator (38), UDP (8), IP (18) and MAC (14 bytes). The *accelerator header* contains a frame number indicating which frame the pixels in this packet belong to and an x and y coordinate indicating the spatial location of the first pixel present in the packet. To support a multiple camera setup[1], a camera identification number indicates the source of the pixels. The accelerator output consists of 4 bytes per pixel which contain the absolute difference, grey value and skin segmented output.

The configuration packets contain parameters values for the accelerator. These parameters determine the operation of several image processing operations present in the accelerator. The configuration packets have the same size as the pixel packets in order to simplify the protocol handling on the PowerPC processor present in the FPGA. The meaningful part of the payload is only 80 bytes. This is, however, not a big issue considering that configuration packets are usually not intermingled with pixel data at run-time. Please refer to appendix B for a complete overview of the pixel packet and configuration packet structure.

[1] A multiple camera setup allows several cameras to be connected to the host PC and might be used for example to improve the accuracy of hand gesture recognition. The accelerator supports such a setup by providing a camera identification number in the accelerator header of the pixel packets.

5.1.3 Accelerator architecture

The architecture of the accelerator is depicted in figure 5.1 and consists of several distinct components[2] interconnected by busses. The following components are present:

- Ethernet media access layer component (EMAC): the EMAC is responsible for receiving and sending of data packets. When an incoming packet is received, a processor interrupt will be raised in order to transfer the packet to the data memory;
- Processor: the processor is responsible for the overall coordination of the components present. Furthermore, it must ensure a timely transfer of data between the components. The processor transfers incoming and outgoing packets from and to the EMAC, and it transfers packets to and from the pixel processing pipeline (PPP) subsystem;
- Data memory: this memory is used as an intermediate buffer between the ethernet component, the processor and the pixel processing pipeline subsystem;
- Instruction memory: this memory contains the program code (instructions) for the PowerPC processor;
- Interrupt controller; the processor contains a single interrupt port but the architecture contains two interrupt sources: the EMAC and the PPP. The interrupt controller is required to multiplex the two interrupt signals into a single signal;
- Pixel processing pipeline subsystem: the PPP is responsible for the image processing operations performed on the pixel data;
- PLB and OPB bus; the Processor Local Bus (PLB) and On-Chip Peripheral Bus (OPB) provide shared communication paths between the different components;
- Bus bridge; the system comprises two bus types: a high-speed (PLB) bus and a low-speed (OPB) bus. The bridge connects these two busses and takes care of the timing;
- PPP Bus Interface (IPIF): provides an interface to the PLB bus in order to simplify the design of the PPP subsystem. It also provides two additional FIFO's for temporal storage.
- Universal Asynchronous Receiver Transmitter (UART); the UART provides the processor with a standard input and standard output for status and debug information to an external computing system.

The interaction of the different components directly involved in processing pixel packets will be illustrated by a simple example of a single receive packet event. The example demonstrates the pixel packet dataflow in the accelerator and is shown in figure 5.2.

The following sequence of actions will be executed upon receiving a pixel packet:

[2]The EMAC, PLB, OPB, Bus Bridge, IPIF, UART and Interrupt controller are standard components provided by Xilinx.

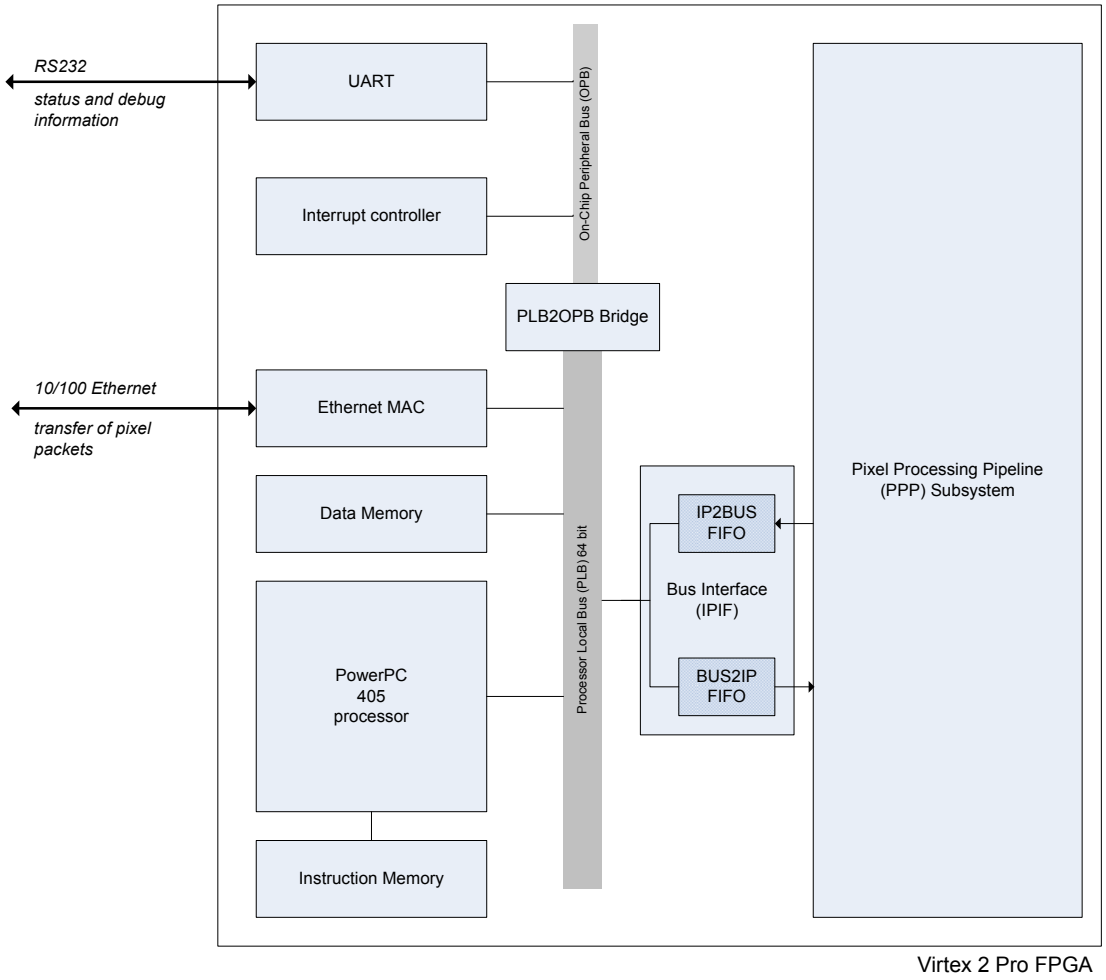

Figure 5.1: system architecture

1. an incoming pixel packet arrives at the EMAC;
2. the EMAC receives the packet in its internal buffer and generates a processor interrupt;
3. the processor starts the interrupt handler which transfers the packet into the data memory and analyzes the packet header;
4. the processor transfers the packet to the pixel processing pipeline input buffer (BUS2IP);
5. the PPP processes the packet on a pixel by pixel basis and stores the results in the PPP output buffer (IP2BUS), and generates an interrupt when a complete pixel packet has been processed;
6. the processor executes the interrupt handler which transfers the packet back into its data memory;

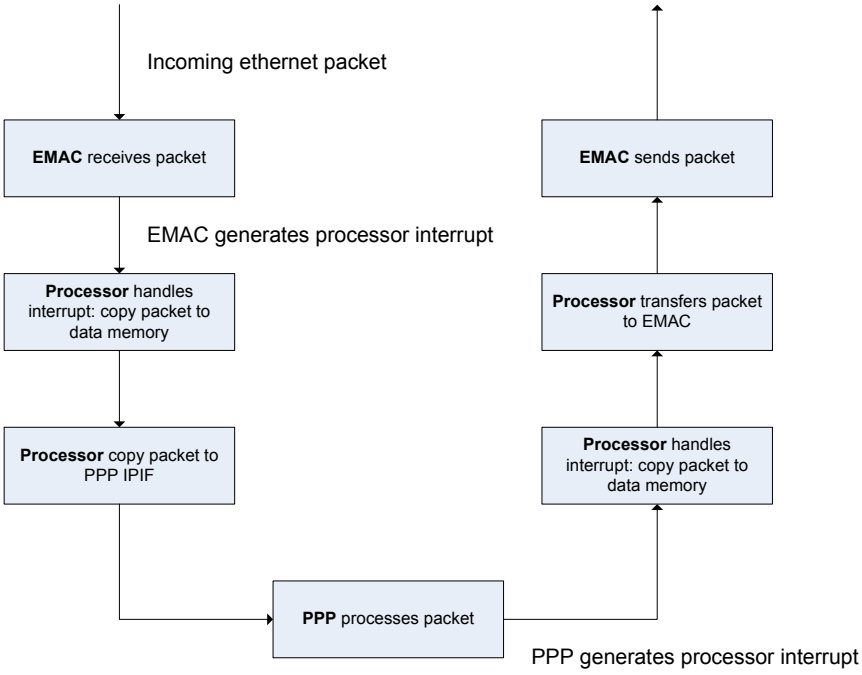

Figure 5.2: example pixel packet flow

7. the processor transfers the packet to the EMAC and orders the EMAC to transmit the packet to the host PC;
8. the EMAC transmits the packet to the host PC using the direct Ethernet connection.

Instead of transferring incoming packets directly to the PPP, the packets are first transferred to the processor data memory. The reason is that the processor should be able to examine the packet headers in order to determine whether the packet should be transferred to the PPP or not. Configuration packets for example must not be transferred to the PPP but interpreted by the processor instead. In addition, there may be network traffic unrelated to the accelerator which need to be discarded.

To resolve temporary and irregular delays in the accelerator, for example caused by two interrupts to be serviced simultaneously, two IPIF FIFO buffers which can contain multiple packets are used.

5.1.4 Pixel Processing Pipeline subsystem architecture

The skin segmentation algorithm as presented in appendix A can be divided 10 dependent stages. Each stage contains a certain amount of different independent operations which can be performed simultaneously.

The PPP subsystem implements the building blocks of this algorithm. We decided to implement the PPP by utilizing a pixel pipelined architecture in order to be able to

process several pixels simultaneously. The amount of pixels which can be processed at a single moment depends on the amount of stages present in the pipeline.

Figure 5.3 shows the architecture of the pixel processing pipeline. The operations indicated correspond to the operations of the skin segmentation algorithm. The dotted lines between the pipeline stages represent pipeline registers which contain the temporary results from the preceding stage. The pipeline controller is responsible for the coordination of the entire processing pipeline. It determines when all pipeline stages have finished processing in order to start the next pipeline cycle.

The image buffer present in the algorithm is used to determine the absolute difference between two consecutive images. It was decided not to implement this buffer in hardware. This would either require a large amount of block RAM or add additional complexity for interfacing external Dual Data Rate RAM. Instead, the host PC is responsible for providing the required pixel values. This can be accomplished if the host PC driver utilizes at least two image buffers. It includes the corresponding pixels of the previous image when transmitting an image to the accelerator.

5.2 Implementation constraints

The accelerator is required to process video frames at 25 frames per second. Each individual video frame has a width of 640 pixels and a height of 480 pixels. Each pixel consists of a red, green and blue color components, each represented using a single byte. Additionally, each pixel contains an 8 bit grey value of the corresponding pixel from the previous frame. Thus each pixel is represented by 32 bits.

5.2.1 Network bandwidth

The decision was made to transfer the video frames to and from the accelerator without data compression. The advantage of uncompressed data is that we do not require both a compression component on the host PC and a decompression component within the accelerator. However, this approach requires a higher amount of bandwidth compared to the case when data compression is used.

It is also important to consider the communication overhead introduced when utilizing the packet based networking communication. The TCP/IP communication standard [26] states that a maximum of 1500 data bytes can be transferred per packet. Additional control and error detection information is appended to the data to allow communication control between the host PC and the accelerator.

At any given video resolutions and frame rate, the following bandwidth requirement approximation holds:

$$BW = depth_{bits\_per\_pixel} * I_{width} * I_{height} * R_{frame} * F_{overhead} \tag{5.1}$$

where the *depth* indicates the amount of bits required to represent a pixel, I_{width} and I_{height} indicate the image dimensions, R indicates the frame rate and F represents the overhead factor caused by the additional header data present in a pixel packet.

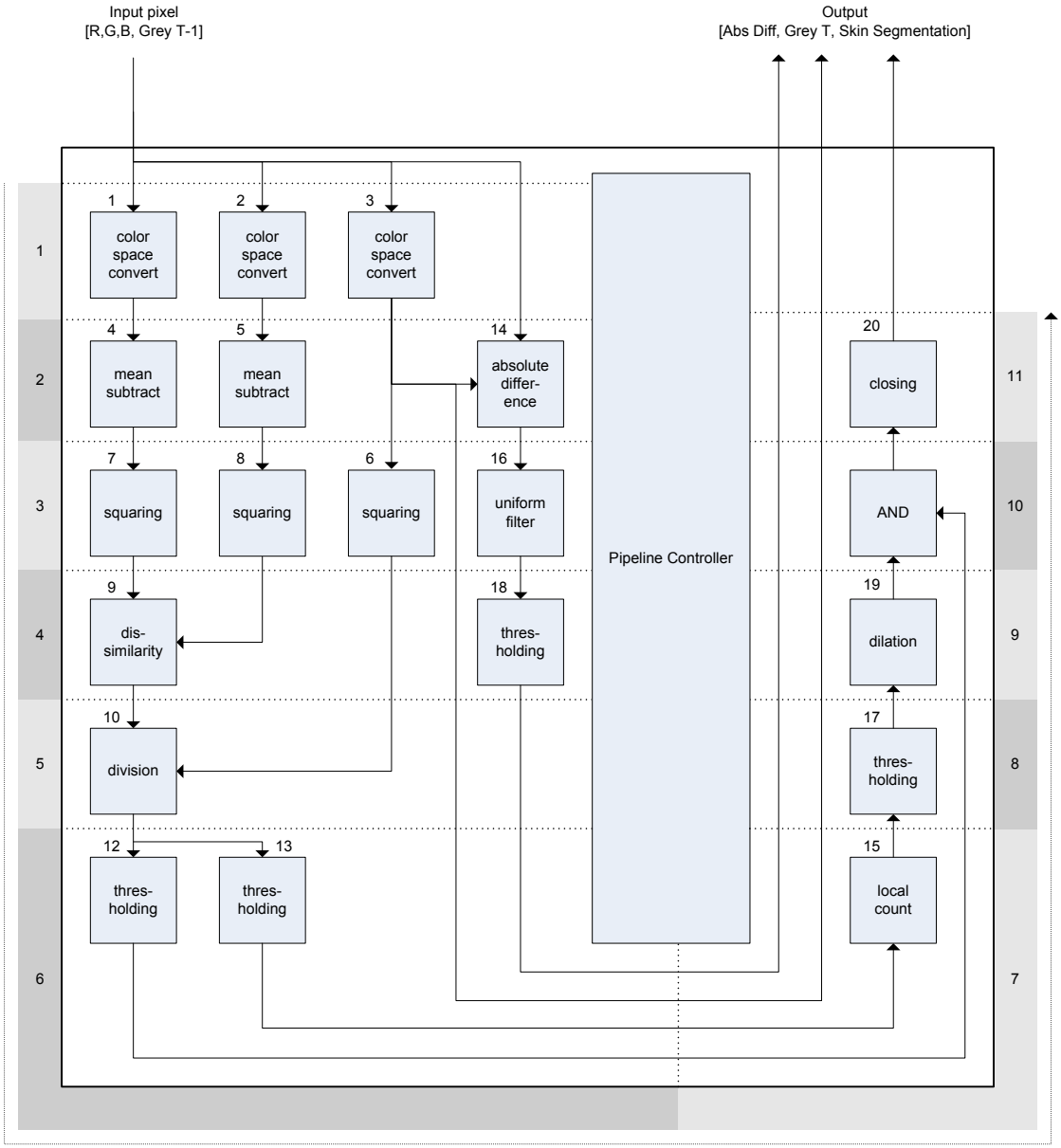

Figure 5.3: pixel processing pipeline architecture

As previously explained, the pixel packets contain 1280 bytes pixel data and 78 additional bytes required for protocol headers (as mentioned in section 5.1.2), resulting in communication overhead factor of approximately 6%.

Given a pixel depth of 32 bits, a frame rate of 25 frames per second and a network packet overhead factor of 6%, the resulting bandwidth requirements are 62,11 Mbps for image dimensions of 320x240, 248,44 Mbps for 640x480 and 636 Mbps for 1024x768 pixels. Since our implementation utilizes a 10/100 Mbps Ethernet MAC, it is limited to image dimensions of 320x240 pixels, however a gigabit Ethernet MAC would enable

image dimensions of either 640x480 or 1024x768.

5.2.2 Accelerator throughput

We decided to utilize communication packets containing 320 pixels per packet. The amount of packets to be processed by the accelerator can thus be estimated as follows:

$$R_{packet} = \frac{I_{width} * I_{height}}{320} * R_{frame} \tag{5.2}$$

where I_{width} and I_{height} denote the image dimensions and R_{frame} indicates the frame rate. With image dimensions of 640x480 pixels and a frame rate of 25 frames per second, the processing rate R_{packet} is 24000 packets per second. Since Ethernet communication allows full duplex operation[3] this effective packet processing rate doubles to 48000 packets per second.

To test the PowerPC software overhead of packet handling, we developed two packet handling approaches. The first approach was based on utilization of a micro kernel[4] with support for processes. We developed processes for receiving packets from the EMAC and the PPP. The second approach involved two interrupt handlers for the same tasks. Through empirical observation we concluded that the process based approach was limited in performance due to the context switching overhead involved. We therefore decided to implement the interrupt based approach.

The interrupt routines are required to finish processing within the time available for a single packet. Assuming the CPU clock frequency is 300 MHz, the following amount of clock cycles are available to each routine:

$$T_{max} = \frac{F_{clock}}{R_{packet}} = \frac{300 * 10^6}{48000} = 6250 \text{ clock cycles} \tag{5.3}$$

where R_{packet} indicates the full duplex packet processing rate and F_{clock} indicates the clock frequency. The PowerPC reference manual [30] states that the typical amount of clock cycles required per instruction is 1. Thus, each routine may consist at maximum of approximately 6250 instructions.

Since the accelerator utilizes a communication bus, it is important to consider the available bus bandwidth. The bus conforms to the Processor Local Bus standard [6] and provides a data transfer width of 64 bits. The PLB bus operates at a maximum frequency of 100 MHz. When utilizing burst transfers [5] the accelerator is effectively able to transfer 64 bits of data per bus clock cycle. Thus, the available bus bandwidth can be approximated as follows:

[3] Full duplex operations allows for simultaneous sending and receiving of data. On the contrary, half duplex only allows for either sending or receiving of data at a given moment.

[4] A micro kernel is a minimal implementation of an operating system. It supports minimal operating system functionality such as thread management and inter-process communication. The micro kernel provided with the FPGA and which was used for our evaluation is called Xilkernel.

[5] Burst transfers consist of a sequence of data transfers initiated by a single command. The effective throughput of burst transfers can be as high as one data transfer per clock cycle. On the contrary, single data transfer require a command for each transfer requested.

$$BW_{plb\_bus} = F_{clock} * W_{bus} = 100 * 10^6 * 64 \approx 6104 \text{ Mbps} \tag{5.4}$$

where F_{clock} denotes the bus clock frequency, W_{bus} denotes the bus width and BW_{plb_bus} denotes the available bus bandwidth. Within the accelerator, a pixel packet will be transferred four times over the bus (EMAC to memory, memory to PPP, PPP to memory and memory to EMAC). Consequently, the total bus bandwidth required can be estimated as follows:

$$BW_{required} = BW_{network\_half\_duplex} * 4 = 248,44 * 4 \approx 994 \text{ Mbps} \tag{5.5}$$

where $BW_{network\_half\_duplex}$ denotes the bandwidth required for sending the pixel packets to the host PC and $BW_{required}$ denotes the required bus bandwidth. Since the available bus bandwidth of 6104 Mbps exceeds the required bandwidth of 994 Mbps, we can conclude that the available bus bandwidth is sufficient to allow for the targeted data rate.

5.2.3 Pixel Processing Pipeline throughput

Given the input video dimensions and frame rate, the following pixel processing rate is required:

$$R_{pixel} = I_{width} * I_{height} * R_{frame} \tag{5.6}$$

where R_{pixel} denotes the pixel processing rate, I_{width} and I_{height} denote the image width and height and R_{frame} denotes the frame rate. For image dimensions of 640x480 pixels and a frame rate of 25 frames per second the pixel processing rate is 7680000 pixels per second.

The following amount of clock cycles is available for each pipeline stage:

$$T_{stage} = \frac{F_{clock}}{R_{pixel}} \tag{5.7}$$

where F_{clock} is the clock frequency and R_{pixel} denotes the pixel processing rate. Assuming a clock frequency of 100 MHz and a pixel processing rate of 7680000 pixels per second, the amount of clock cycles available for each pipeline stage is 13.

5.3 Conclusion

The system architecture discussed in this chapter is expected to allow the real-time processing of input pixels. It is based on an FPGA which is connected to a host PC using a direct Ethernet connection.

The host PC initially sends a configuration packet containing image processing parameters to the accelerator. After initialization, the host PC will send a continuous stream of pixel packets to the accelerator. The accelerator processes these packets and returns a stream of output pixel packets.

The main components of the accelerator are the PowerPC processor and the Pixel Processing Pipeline. The processor is responsible for the network communication and the

overall system control, the pixel processing pipeline is responsible for the implementation of the skin segmentation algorithm. The implementation of the pixel processing pipeline remains fixed during run-time. However, it can be modified by adapting the source code of the algorithm and resynthesis of the design. The parameters of the skin segmentation algorithm can be updated when the system is operational by sending dedicated configuration packets to the accelerator.

Since the selected development board contains a 100 Mbps Ethernet network connection, an image resolution of 320x240 pixels can be supported at a frame rate of 25 frames per second. A gigabit Ethernet connection would allow image resolutions of 640x480 pixels at the same frame rate.

6 Implementation

The main components of the architecture of the accelerator are the PowerPC processor and the Pixel Processing Pipeline. Communication between host PC and the accelerator occurs via direct network connection. This chapter discusses the implementation of the accelerator architecture and the pixel processing pipeline. The architecture provides a generic framework for real-time image processing, whereas the pixel processing pipeline implements the specific components for the skin segmentation algorithm. The first section provides details on the implementation of the processor functionality, the second section discusses the pixel processing pipeline. Details on the timing and resource utilization of the PPP components are available in appendix D.

6.1 Processor functionality

The PowerPC processor is responsible for the overall coordination of the accelerator. It implements the functions required by the network protocol and controls the transfer of pixel packets between the EMAC and the PPP. The following subsections provide details on the network protocol and packet handling.

6.1.1 Network protocol selection

To establish a reliable communication link between the accelerator and the host PC, the processor should utilize the most suitable network protocol for the purpose. The Open System Interconnection (OSI) model for network communication [31] defines a set of layers that are involved in the communication between network connected systems. Generally, different software layers present within a system correspond to the different OSI layers and implement the required functionality.

To minimize the network protocol overhead, we chose to use the User Datagram Protocol (UDP) [25] instead of the Transmission Control Protocol (TCP). The latter protocol has significant control overhead in order to guarantee delivery and reception of correct data packets. This might even result in packet retransmission when these have been damaged during transmission. The former protocol requires less overhead but does not guarantee delivery and reception of correct data packets. However, judging by the real-time objective of our accelerator and the video application being used, we prefer damaged packets over packet retransmissions.

The performance issues encountered when developing high-speed communication applications have been discussed in [5], [10] and [16]. The bottlenecks discussed in these papers are caused by the memory copy, checksum calculation and interrupt overhead introduced by the network protocol handling. In [5] it is shown that several methods can be used in order to increase the communication throughput. Several alternative

network protocols ([21], [20]) have been proposed to reduce the processing overhead. However, these proposals have not yet been widely implemented and thus will result in compatibility problems if used in this project.

Our PowerPC software implementation uses the zero memory copy method, this effectively reduces the amount of packet transfers between software layers. Furthermore, our implementation omits the checksum calculation phase since this is allowed when using the UDP protocol. This prevents the host PC to detect errors in a packet but results in a significant communication performance improvement since checksum calculations involves all data bytes in the packet.

6.1.2 Interrupt handling

The second task of the processor is coordination of the transfer of pixel packets between the EMAC, data memory and the PPP. To minimize the transfer overhead, it was decided to implement this functionality using interrupt routines and direct memory access (DMA).

Currently two types of interrupts have been implemented: the *EMAC receive packet interrupt* and the *PPP packet processed interrupt*. The former will be generated by the EMAC whenever a network packet has been received, the latter when the PPP has finished processing a pixel packet.

The interrupt handling routine for the *EMAC receive packet interrupt* is responsible for transferring the packet from the EMAC component to the data memory. It also needs to identify the type of packet and process it accordingly. Currently the following types of packets are recognized:

- Network protocol packets; both IP protocol and UDP protocol packets are processed. Furthermore, ICMP ("ping") packets are recognized to provide the possibility for testing the communication link between the accelerator and the host PC. TCP and other network protocol packets will currently be discarded.

- "Hello" packets; these packets will be send by the host PC in order to inform the accelerator of the IP address and port number of the host PC. This information is required by the accelerator to return the processed pixel packets.

- Configuration packets; these packets contain parameter values for the PPP components. Upon receiving a configuration packet, the processor writes the appropriate values to the programmable registers of the PPP.

- Pixel packets; these packets contain pixels which will be transferred to the PPP in order to be processed.

The interrupt handling routine for the *PPP packet processed interrupt* first transfers the packets to the data memory. Then it appends the required protocol headers and finally transfers the packets to the host PC.

To reduce the per-packet processing time, it was decided that the processor should not be responsible for the administration of pixel packet header data. Consequently, the processor only transfers the packets to and from the PPP instead of also having to match

each processed pixel packet with the corresponding header. Thus, this responsibility is delegated to the PPP by simply transferring the header data along with the pixel values into the PPP.

As mentioned earlier, transfer of the pixel packets between the EMAC, system memory and the PPP uses DMA. This method allows data transfer at a higher throughput compared to programmed input/output[1] (PIO) data transfer. A DMA operation transfers data from a specified source address to a destination address. Since the PLB bus has a width of 64 bits, a DMA operation effectively transfers 64 bits per clock cycle. It is required that both source and destination address are aligned on a 64 bit address, thus the processor needs to align the source and destination memory sections at such an address.

6.2 Pixel Processing Pipeline

This section discusses the implementation of our pixel processing pipeline. First, the implementation of the pipeline architecture is presented. Next, the protocol used in the PPP is discussed. Then the color space conversion, erosion and dilation, uniform filter and local count components are discussed.

6.2.1 Pipeline implementation

The implementation diagram of the pixel processing pipeline is depicted in figure 6.1. In addition to the components mentioned in section 5.1.4, figure 6.1 shows the following components: input multiplexer, header bypass fifo, pipeline registers, delaying pipeline registers and output demultiplexer.

The multiplexer and demultiplexer are required since a single PLB bus transfer contains 2 pixel values. The input multiplexer at the first pipeline stage therefore splits a 64 bit value into two subsequent pixel values. The output demultiplexer in the last pipeline stage then concatenates two subsequent processed pixel values into a single 64 bit value.

Since the PPP is responsible for matching the processed pixel values to their corresponding header, a header bypass fifo is present. The fifo depth depends on the amount of pixel packets that can be present in the pipeline, and is determined by the amount of pipeline stages and the initial latency of the local operations. These local operations require several image lines to be present in their local buffers, and thus will contain several pixel packets. A 5x5 local operation has an initial delay of approximately 2 image lines, a 7x7 local operation approximately 3 image lines. Our implementation uses two pipeline stages with 5x5 local operations and one pipeline stage with a 7x7 local operation. The header fifo should therefore be able to accommodate at least 7 headers for 320x240 and 14 for 640x480.

At the start of a new pixel packet, the input multiplexer transfers the header data to the header fifo. The output multiplexer then reads the header data from the header fifo at the start of a new processed pixel packet. Both input multiplexer and output demultiplexer use a current pixel counter to identify the start of a new pixel packet.

[1]Programmed input/output denotes transfer of data involving processor instructions for each data unit.

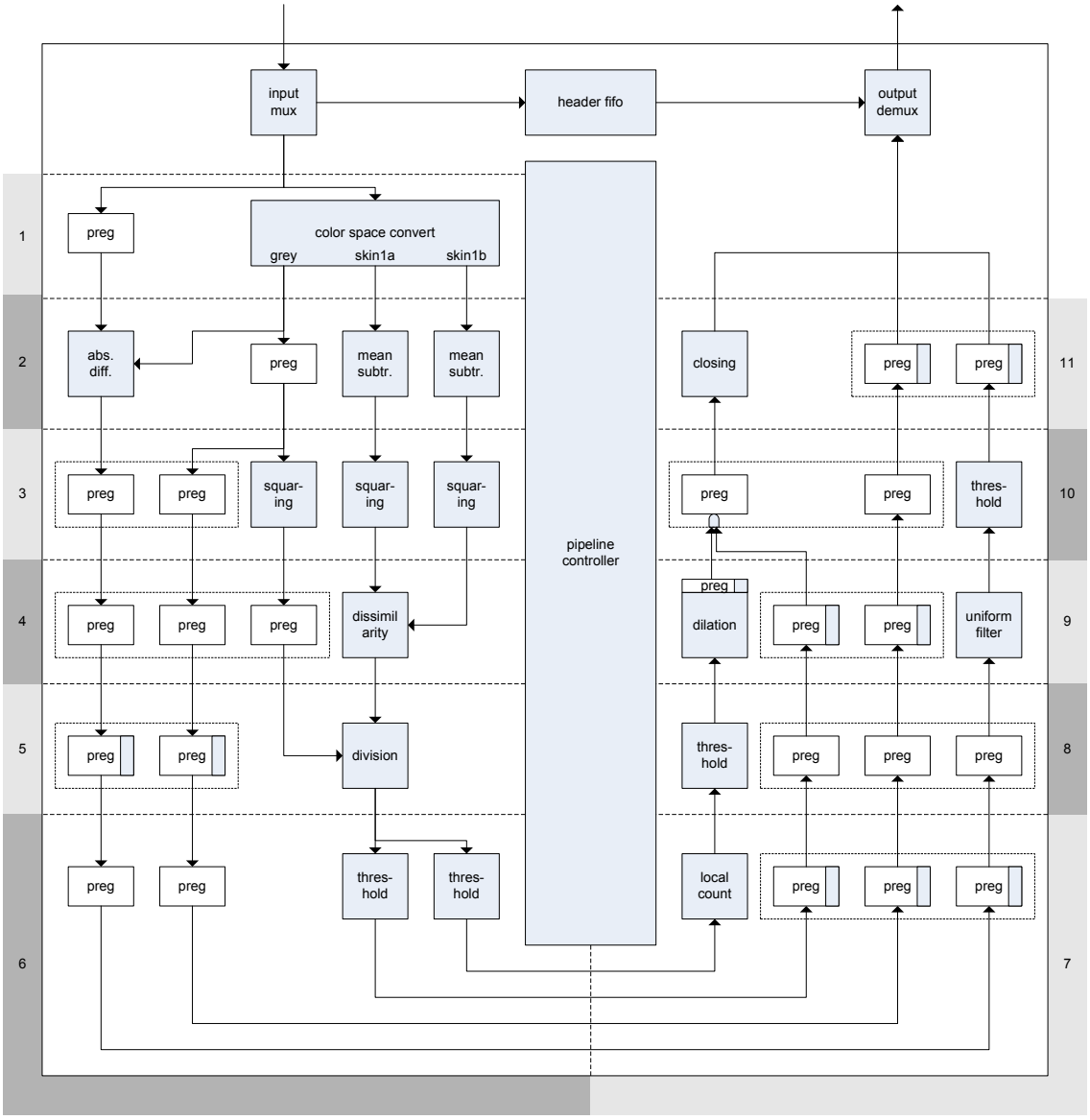

Figure 6.1: PPP implementation diagram

The operation of several components within the PPP can be configured at run-time by modifying their respective parameters. These parameters are accessible by the processor through the PLB slave registers of the PPP. For example, a certain register determines the threshold value of the absolute difference calculation. Please refer to appendix C for a detailed overview of these registers.

6.2.2 Pipeline protocol

Each component is required to provide a set of interfaces for correct operation of the pipeline. It was therefore decided to develop a simple pipeline protocol to which each

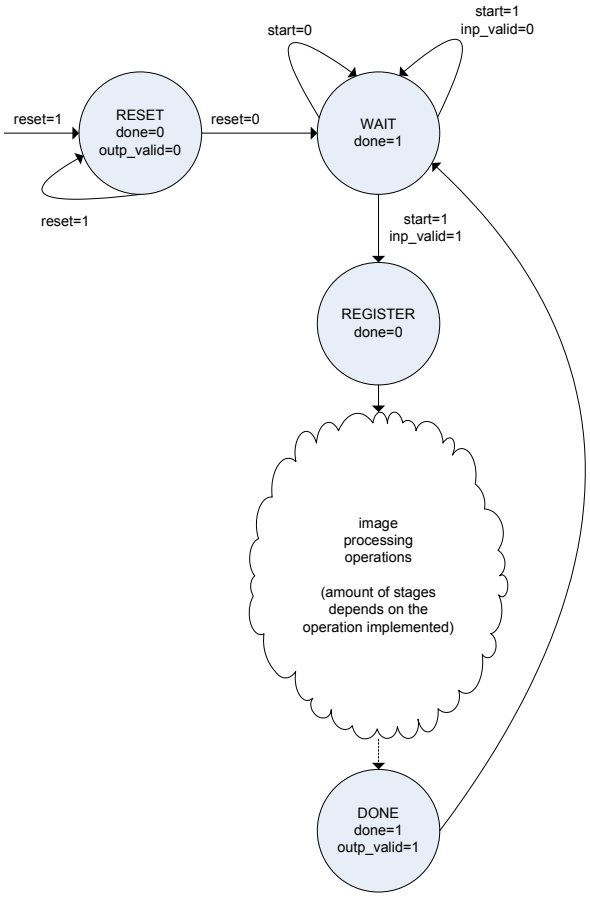

Figure 6.2: pipeline protocol state diagram

component should adhere. The pipeline protocol can best be described by the state diagram as depicted in figure 6.2 and can be described as follows. The *done* signals of all pipeline components are connected to the pipeline controller. This controller issues a *start* signal when all components have finished processing to indicate the start of a new pipeline cycle. A component then starts processing if the *valid* input is set at its input. It is required that the component first copies its input operands before starting the actual processing. When the component is ready, it sets its *done* output. If the *valid* input of the component was set at the start of the pipeline cycle, its output will also be valid at the end of the pipeline cycle. In that case the component needs to set its *valid* output. If the *valid* input of the component was reset at the start of the pipeline cycle, then the component is required to reset its *valid* output. This mechanism is required to keep the pipeline operational in case of input buffer underrun.

This protocol has been implemented in the PPP components. The following sections will therefore omit the discussions of the protocol related functions and concentrate on the implemented operations.

6.2.3 Color space conversion

The color space conversion block (present in pipeline stage 1) implements the vector by matrix multiplication as depicted in equation 6.1.

$$\begin{bmatrix} Grey & Skin1a & Skin1b \end{bmatrix} = \begin{bmatrix} R & G & B \end{bmatrix} \begin{bmatrix} W_{greyR} & W_{skin1aR} & W_{skin1bR} \\ W_{greyG} & W_{skin1aG} & W_{skin1bG} \\ W_{greyB} & W_{skin1aB} & W_{skin1bB} \end{bmatrix} \quad (6.1)$$

where the output $Grey$ contains the grey value representation of the input pixel, the $Skin1a$ and $Skin1b$ outputs represent the components of the skin model. Since the input pixel is represented using a red R, green G and blue B color component, the weights matrix contains conversion weights for each component.

Equation 6.1 consists of 9 multiplications and 6 additions. This can be concluded when rewriting the individual conversion equations. For example, the grey value is calculated as shown in equation 6.2

$$Grey = R * W_{greyR} + G * W_{greyG} + B * W_{greyB} \quad (6.2)$$

where R, G, and B represent the input pixel color components and W_{greyR}, W_{greyG} and W_{greyB} represent the conversion weights for the grey scale conversion.

The input pixel components are multiplied with the corresponding weights and the results are stored in three separate registers. The conversion weights matrix has been implemented using programmable registers which can be modified at run-time.

The architecture of the color space conversion block is depicted in figure 6.3. As can be seen in the figure, a single multiply-accumulate (MAC) unit is used. The MAC multiplies two input operands and accumulates the results of the subsequent operations. The implementation of the MAC uses the embedded hardware multipliers in the FPGA to implement the multiplication. The controller is responsible for selecting the color and weight components and storing the result in the appropriate output registers.

The three color components of the input pixels are represented as unsigned numbers using 8 bits, ranging from 0 to 255. The conversion weights have been implemented using 9 bits in 2's complement format, ranging from -256 to 255. The color components widths are extended to 9 bits (by appending a zero) to match the conversion weights width. The color components and conversion weights are subsequently multiplexed into the operand inputs of the MAC by the controller. The controller uses the *First Data* (FD) and *Next Data* (ND) MAC inputs and *Ready* (RDY) and *Ready For Data* (RFD) to control the operand loading and result storing process.

6.2.4 Neighborhood implementation

The uniform filter, local count, dilation and closing components present in the PPP implement a 5x5 neighborhood operation. The output for these operations is determined by examining the neighborhood of the corresponding input pixel. The neighborhood operation scans the input image pixels with a kernel window on a row basis, starting at

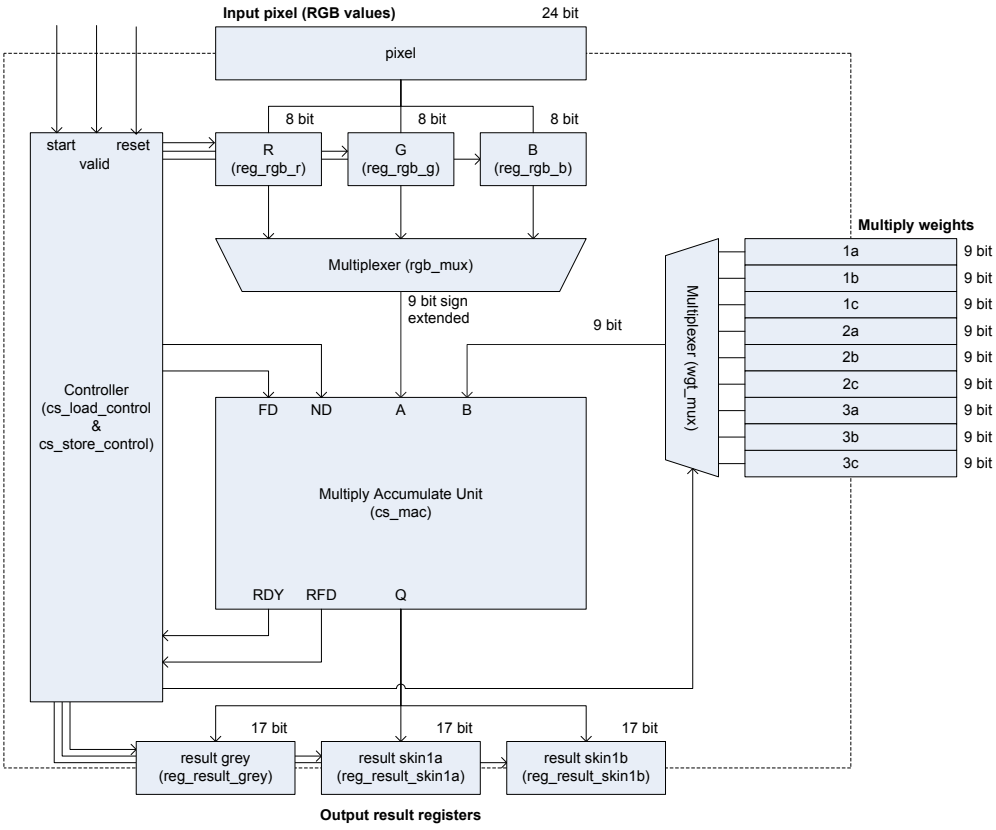

Figure 6.3: color space conversion block diagram

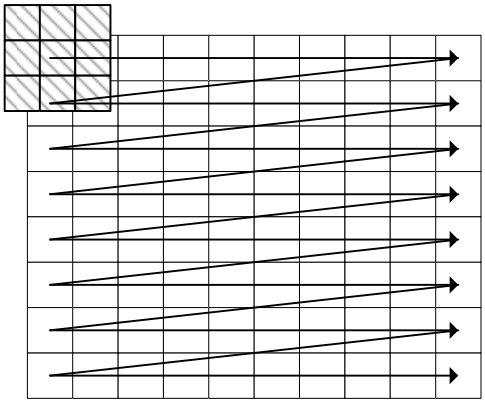

Figure 6.4: kernel scanning process

the first row (top left) as depicted in figure 6.4. Please note that the actual operation depends on the implementation of the specific kernel logic and arithmetic.

Figure 6.5 depicts the implementation of the general 5x5 neighborhood operation.

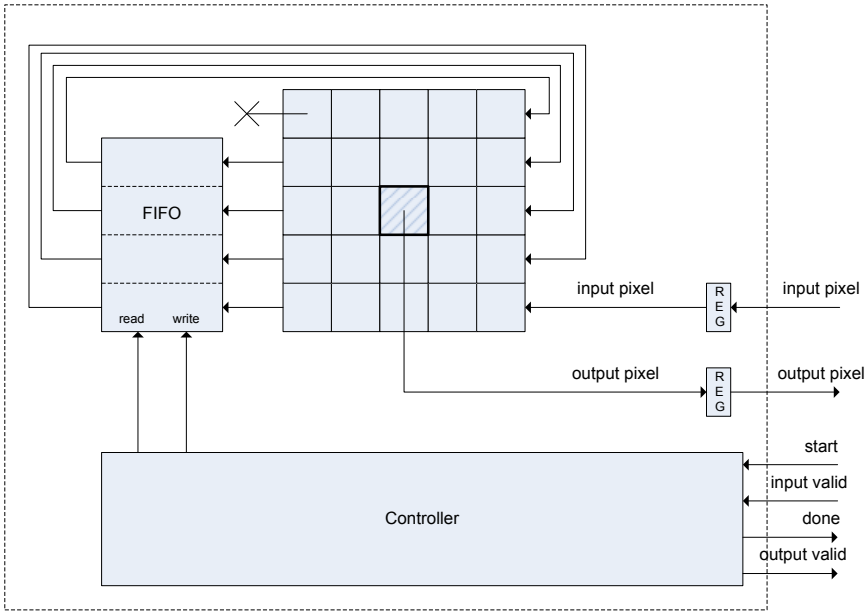

Figure 6.5: general structure of a 5x5 neighborhood operation

To implement the scanning behavior of the kernel window, pixels enter the window at the bottom right. At the start of each pipeline cycle, all pixels in the window are shifted one position left. Pixels being shifted outside the kernel window enter the local 4 channel FIFO. After shifting one image row, these pixels are read from the fifo and reenter the kernel window at the right shifted one row upwards. This effectively implements the scanning process as depicted in figure 6.4. Note that since the kernel window logic depends on the operation implemented, this logic is not shown in this figure.

The controller implements the pipeline protocol handling and additionally controls the read and write signals of the FIFO. Initially, the first pixel of an image enters at the bottom right position of the kernel. However, the output starts whenever the first pixel is located in the center position of the kernel window. This *initial latency* has been implemented by qualifying the output *valid* signal with a pixel counter.

The image boundary conditions represent an additional complexity and occur when the kernel window should include non-existing "pixels" outside the actual input image. The implementation should account for these conditions by either replicating the nearest existing pixels or setting the appropriate values to zero. In order to simplify the implementation it was decided to set these pixel values to zero. This also prevents the algorithm to detect false results at the border areas.

The binary erosion, dilation and closing operations and the uniform filter and local count operations are all based on the above window scanning process. The following sections therefore concentrate on the kernel logic implemented for these operations.

6.2.5 Binary erosion, dilation and closing

For binary erosion and dilation (present in pipeline stage 9), the kernel logic remains quite simple. The implementation of the dilation operation utilizes the logical OR function of the entire neighborhood as shown in equation 6.3.

$$Output = W_{11} + W_{12} + W_{13} + W_{14} + W_{15} + ... + W_{55} \tag{6.3}$$

where the pixels in the neighborhood of the input pixel are denoted as W_{11} for the top left and W_{55} for the bottom right. The input pixel is represented with W_{33}.

Similarly, the implementation of the erosion operation uses the logical AND of the entire neighborhood as shown in equation 6.4.

$$Output = W_{11} \bullet W_{12} \bullet W_{13} \bullet W_{14} \bullet W_{15} \bullet ... \bullet W_{55} \tag{6.4}$$

where the pixels in the neighborhood of the input pixel are denoted as W_{11} for the top left and W_{55} for the bottom right. The input pixel is represented with W_{33}.

Since the binary closing operation is defined as a binary dilation followed by a binary erosion, the closing operation (present in pipeline stage 11) is implemented by cascading instances of a dilation operation and an erosion operation. Similarly, the implementation of a binary opening operation can be similarly achieved by cascading an erosion operation and a dilation operation.

6.2.6 Convolution filter

The implementation of the convolution filter consists of a 5x5 neighborhood operation with parameterizable kernel weights. The current implementation is restricted to positive kernel weights of powers[2] of 2. The motivation is that in this case the kernel operation can be simplified to shifting instead of multiplying the pixel values. However, an advanced implementation using multipliers allows use of arbitrary kernel weights but increases the operation time and resource utilization.

The structure of the convolution filter kernel implementation is depicted in figure 6.6. It was decided to split the operation into three subsequent stages:

1. First stage: neighborhood pixel shifting (multiplication); shifts each neighboring pixel left by the amount of bits specified by the kernel weight. This is depicted in the figure by kernel window blocks with a pixel value input and a shifting amount input. The actual shifting process is not shown but is assumed to take place in these blocks.

2. Second stage: row wise accumulation; accumulates the results of the previous stage in a row wise fashion. The outputs of the kernel window blocks in each row form the inputs of the row accumulators.

[2]The current implementation allows for the following kernel weights: 0, 1, 2, 4, 8, 16, 32, 64, 128 and 256.

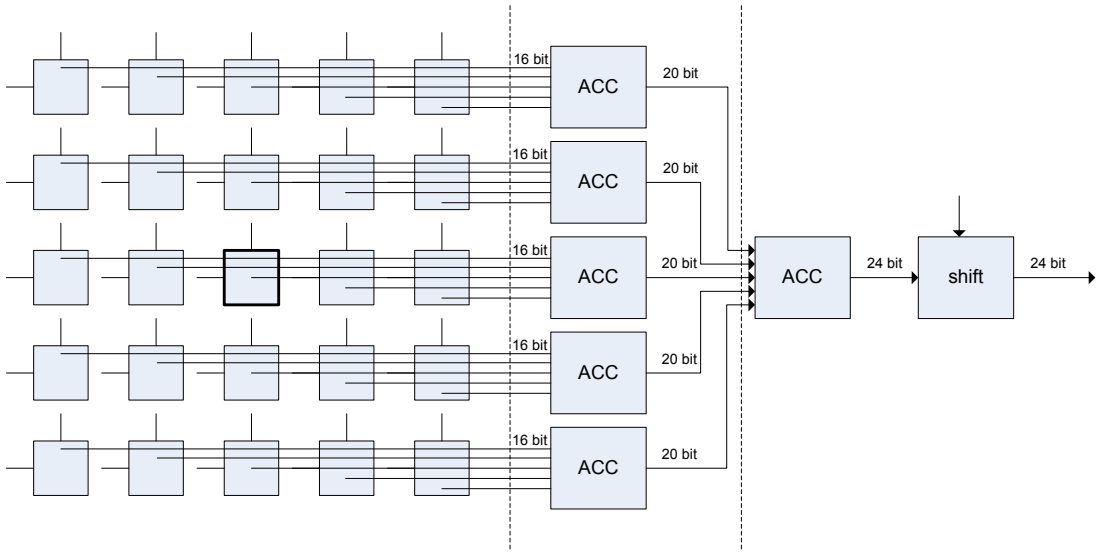

Figure 6.6: (uniform) filter kernel implementation

3. Third stage: column wise accumulation and result shifting; accumulates the results of the previous stage and shifts the result right by the amount of bits specified by a configurable parameter. This allows a division of the result by a power of 2.

With this architecture the *uniform* filter (present in pipeline stage 9) can easily be implemented by utilizing equal weights throughout the kernel. Furthermore, various other filter types can be implemented by utilizing different filter weights and shifting the output if necessary. For example, an approximate implementation of a gaussian filter might be achieved by applying the appropriate filter weights.

6.2.7 Local count

The local count operation (present in pipeline stage 7) consists of a 7x7 neighborhood operation, its structure is similar to the 5x5 neighborhood operation as discussed in section 6.2.4.

The kernel implementation structure is depicted in figure 6.7 and consists of 7 7-input lookup tables connected to a shared accumulator. The pixels of each window row form the inputs to the corresponding one count lookup table. This lookup table contains precalculated one count values for all possible input combinations. Since there are 7 inputs, $2^7 = 128$ possible input combinations are encoded in the lookup table. The outputs of the lookup tables are subsequently multiplexed into the operand input of the accumulator. At the end of every processing cycle the accumulator contains the amount of ones present in the neighborhood.

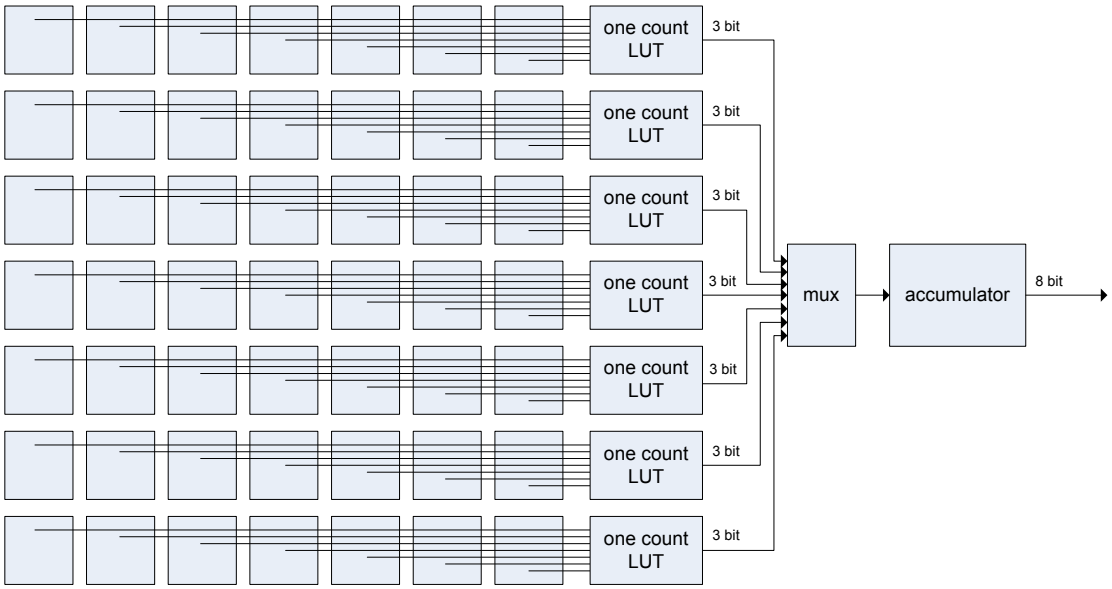

Figure 6.7: local count filter kernel implementation

6.2.8 Remaining operations

The remaining processing blocks (addition/subtraction, multiplication, division and threshold) will not be discussed in detail since these basically consist of standard mathematical operations extended with logic required by the pipeline protocol.

The addition, substraction, multiplication and division components are based on functionality provided by the Xilinx Core Generator. This utility automatically generates an efficient implementation of the respective operations in hardware. The threshold components were similarly realized using standard greater or equal comparators.

The absolute difference component implements equation 3.11. It consists of a subtractor with additional operand swapping logic. This logic swaps the input operands of the subtractor to guarantee that the first operand always contains the largest operand. Since the operands are positive (both represent grey scale values), the result will be positive and thus represent the absolute difference between the two operands.

As mentioned in section 6.2.4, the uniform filter, local count and erosion and dilation components introduce an initial latency. The pipeline registers in the same stage therefore also need to introduce the exact same latency to produce synchronized output. This is required to guarantee synchronized output of the different components in each stage. For this purpose, the delaying pipeline "register" was developed. This register consists of a FIFO to compensate for the initial delay introduced by the components mentioned. It only outputs a value if the other components residing in the same pipeline stage produce a valid output.

Table 6.1: XC2VP30 utilization summary

	used	total	ratio
Slices	12,141	13,696	88%
Slice registers	11,045	27,392	40%
4-input LUTs	17,459	27,392	63%
Block RAM	124	136	91%

6.2.9 Implementation results

The accelerator architecture, the pixel processing pipeline and the image processing components have been implemented using Xilinx Platform Studio 7.1i and Xilinx Integrated Software Environment 7.1i and synthesized using Xilinx Synthesis Tool. The accelerator project files, including the PowerPC software source code and pixel processing pipeline VHDL files, can be found on the accompanying CDROM. Appendix E provides an overview of the files on this CDROM.

A summary of the FPGA resources used by the accelerator is shown in table 6.1. The total utilization of the logic resources (slices) is 88% while only 37% are required for the implementation of the pixel processing pipeline. The remaining resources are used by the accelerator architecture. The high utilization of block RAM is due to the instruction and data memory space needed for the PowerPC processor. A detailed discussion of the resource utilization of the individual accelerator components is available in appendix D.

6.3 Summary

In the chapter we presented the implementation of both the processor functionality and the pixel processing pipeline and its components. The implementation method allows for modification of the operations in next versions of the algorithm. The parameters of the operations can be modified at run-time by updating the parameter registers of the pixel processing pipeline.

System verification and evaluation

7

The component implementations as discussed in chapter 6 have been evaluated and tested in order to verify their correctness. This chapter presents the verification and test results of these components. First, the test results of the individual components are discussed, then the pixel processing pipeline verification results and the overall system test results are presented.

7.1 Image processing components

The image processing components were individually verified before integrating them in the complete pixel processing pipeline. The verification methodology involved comparison of the hardware generated output with software generated output. The simulated hardware output has been achieved through behavioral simulation[1] of the components and writing the output to a text file. The software output was generated by implementing the operations in software using the OpenCV [17] image processing library and writing the output to a text file. The resulting text files were read by a comparison program and the differences were written to another text file. The simulation and software outputs and the differences were converted into images to allow visible inspection. The following subsections discuss the verification results of the components.

7.1.1 Color space conversion

Figure 7.1 depicts the results from the color space conversion. The source RGB image (a) has been converted to grey scale by software (b) and by simulation of the color space conversion component (c) based on equation 3.8. Differences between software and hardware simulation are shown in (d). It should be noted that the differences have been scaled by a factor of 50 to improve visibility, the grey areas therefore represent a difference in pixel values of 1 between the hardware and the software.

The figure shows that in certain areas differences are present. These differences can be attributed to the error in the calculation method utilized by the color space conversion component. For grey scale conversion the following calculation is performed:

$$Grey = \frac{R * 85 + G * 85 + B * 85}{256} \tag{7.1}$$

where $Grey$ is the resulting grey scale output, R, G and B represent the input pixel color components. The grey scale conversion averages the color components by multiplying

[1] Using a hardware description language, components are developed by describing their behavior at the level of clock cycles. Simulation at this level is denoted behavioral simulation. We used the Modelsim SE 6.0a software package for behavioral simulation.

Figure 7.1: color space conversion results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

each component with a factor of 1/3. In our implementation we approximated this factor by using a factor of $85/256 \approx 1/3$. First, the individual color components (R, G, and B) are multiplied by 85 and the results are accumulated. Then, an 8 bit right shift operation, which corresponds to a division by 256, is performed to yield the grey scale conversion result $Grey$. It can therefore be seen that the result is an approximation with the following error factor:

$$\Delta E = 1 - \frac{3 * 85}{256} = \frac{1}{256} \tag{7.2}$$

where ΔE is the error factor. This error factor becomes notable in the higher intensity areas of the image as can be seen in figure 7.1(d). The error in higher intensities is larger and thus results in differences because of truncation in hardware.

7.1.2 Binary dilation, erosion and closing

The dilation, erosion and closing components have been tested using a binary input image. Figures 7.2, 7.3 and 7.4 depict the respective simulation results. The input images are depicted in (a), the software generated outputs in (b), the hardware simulation outputs in (c) and the differences between the software output and simulation output are depicted in the (d) sections of these figures.

The dilation results in figure 7.2 do not show differences between the software output and the hardware output. We can therefore conclude that the hardware produces the

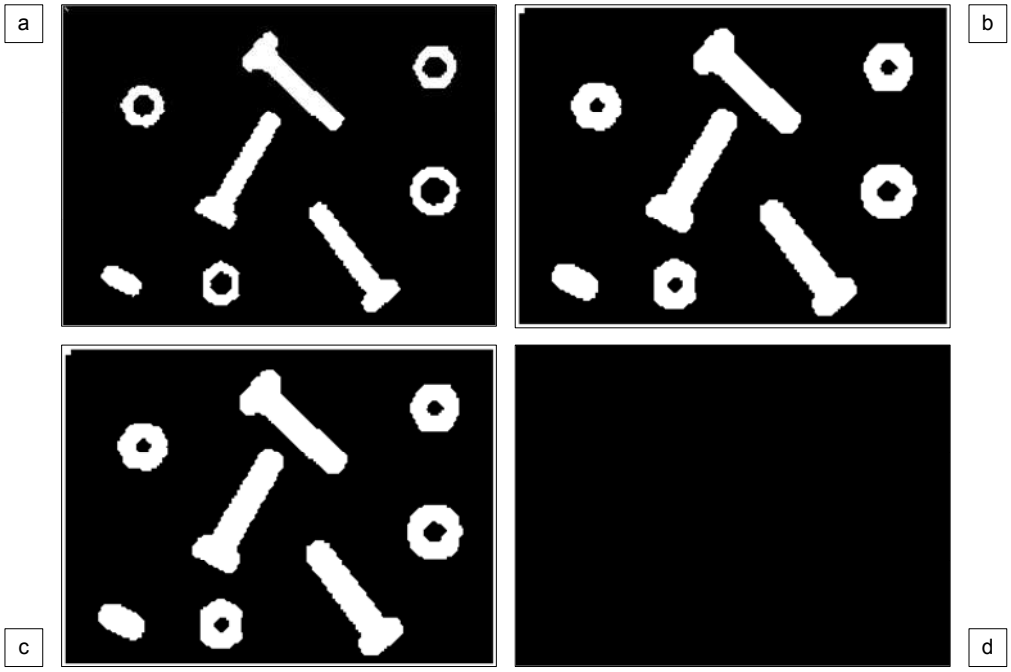

Figure 7.2: binary dilation results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

correct results.

The erosion results in figure 7.3 do not show differences between the software output and the hardware output. However, as mentioned in section 6.2.4, the PPP components assume a value of 0 at the image borders whereas software algorithms often replicate the nearest border pixels. This might result in differences between the software output and the hardware output. It occurs in the closing results depicted in figure 7.4. Since the implementation of the erosion consists of the logical AND of all pixels in the neighborhood, it will always generate output of zero at the border areas. The software output however replicates the border pixels and thus generates a white output at the border areas. For our application we consider this acceptable since the border areas of the input image are of less interest.

7.1.3 Uniform filter

The implementation of the uniform filter was generalized into a convolution operation for 8 bit pixel values with a configurable 5x5 kernel. Our implementation of the uniform filter uses weights of 1. The results of the simulation are shown in the figure 7.5.

Apart from the border areas, the hardware simulation produces correct output. The differences shown in (d) between the hardware simulation output (c) and the software generated output (b) indicates differences in the border areas of the image. The reason can be attributed to the fact that the PPP component handles the border situations differently. Whereas the software algorithm replicates the input pixel values outside the

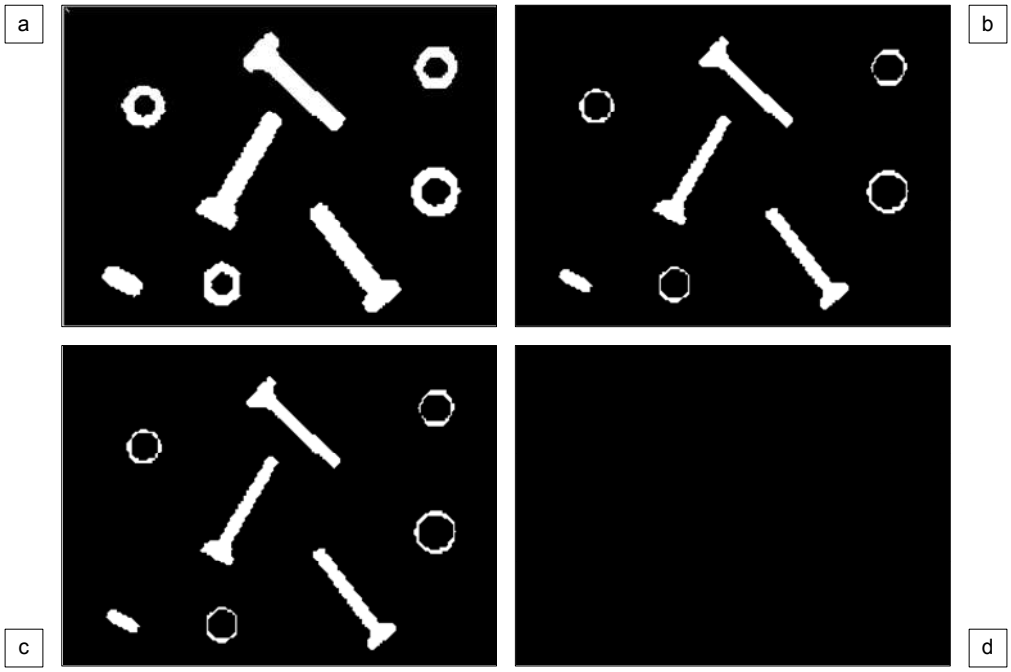

Figure 7.3: binary erosion results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

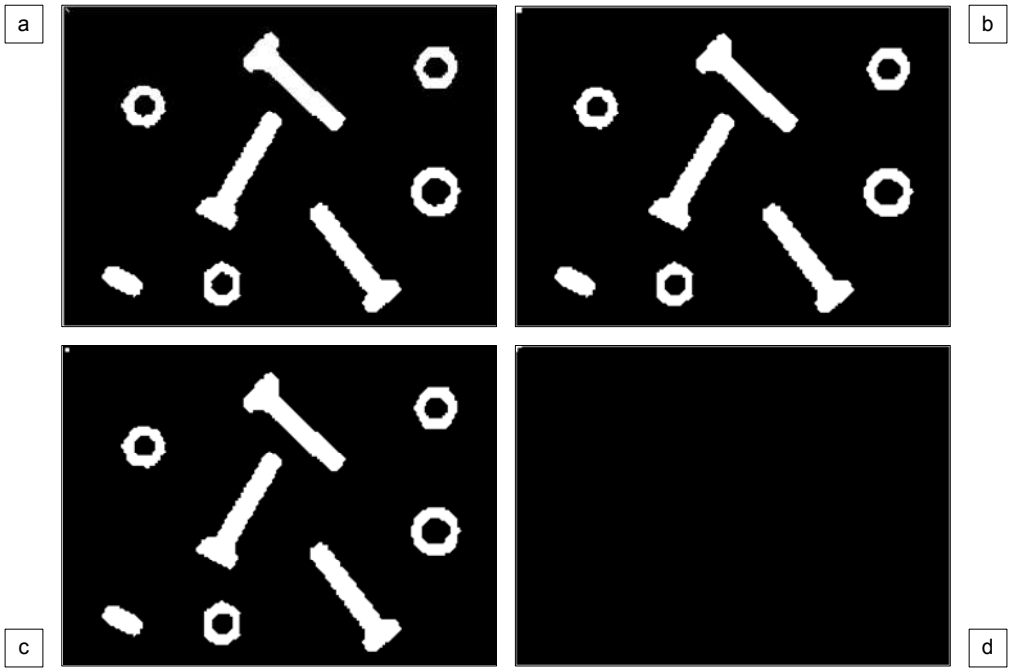

Figure 7.4: binary closing results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

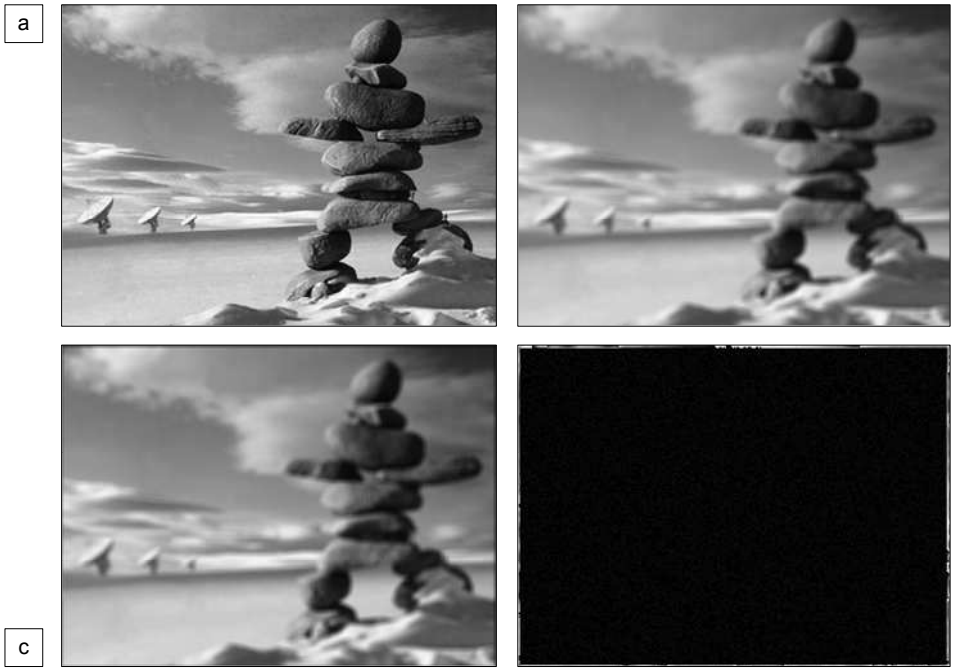

Figure 7.5: uniform filter results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

borders, the PPP component sets these values to 0.

7.1.4 Local count

Simulation results for the local count operation are depicted in figure 7.6. The output of both software (b) and hardware simulation (c) have been increased by a factor of 5 in order to aid visibility.

Apart from the border areas, the hardware simulation produces correct output. Also for this component the border areas differ with the software implementation because of the fact that the component assumes values of 0 near these boundaries. In our implementation this results in darker borders in the output image.

7.1.5 Remaining operations

The simulation results for the threshold operation can be found in figure 7.7. The threshold value was set to 200, the results show that the hardware simulation (c) produces output identical to the software implementation (b) since there are no differences visible in (d).

The remaining operations present in the PPP have been verified through testing of all possible input combinations. With the exception of the division component, all components generate flawless results. The division component results in minor truncation errors because of the fixed size of the fractional remainder. Since the implementation

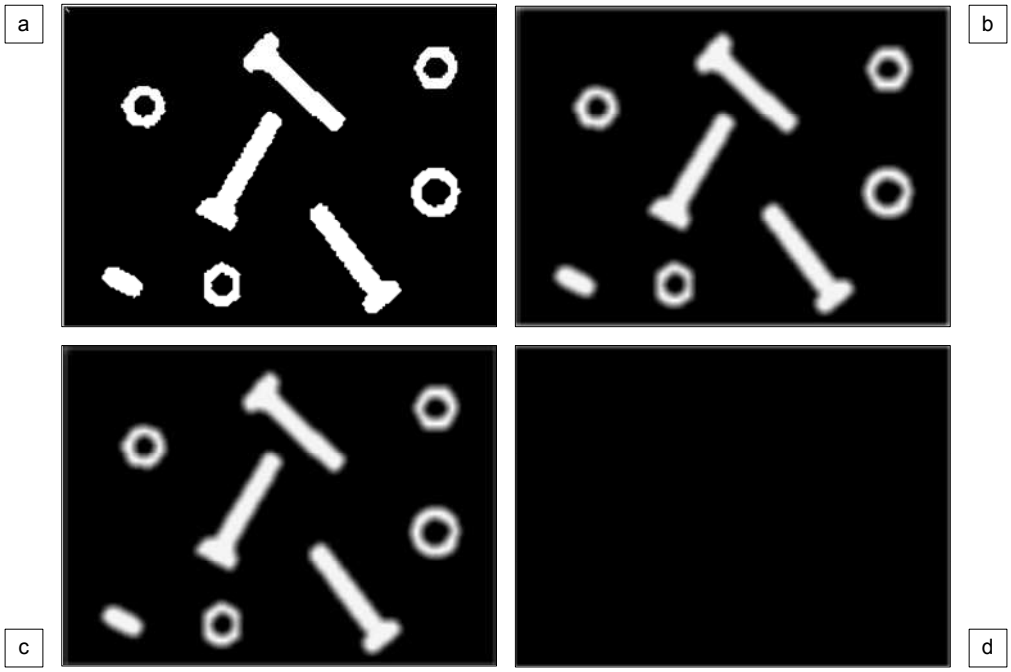

Figure 7.6: local count results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

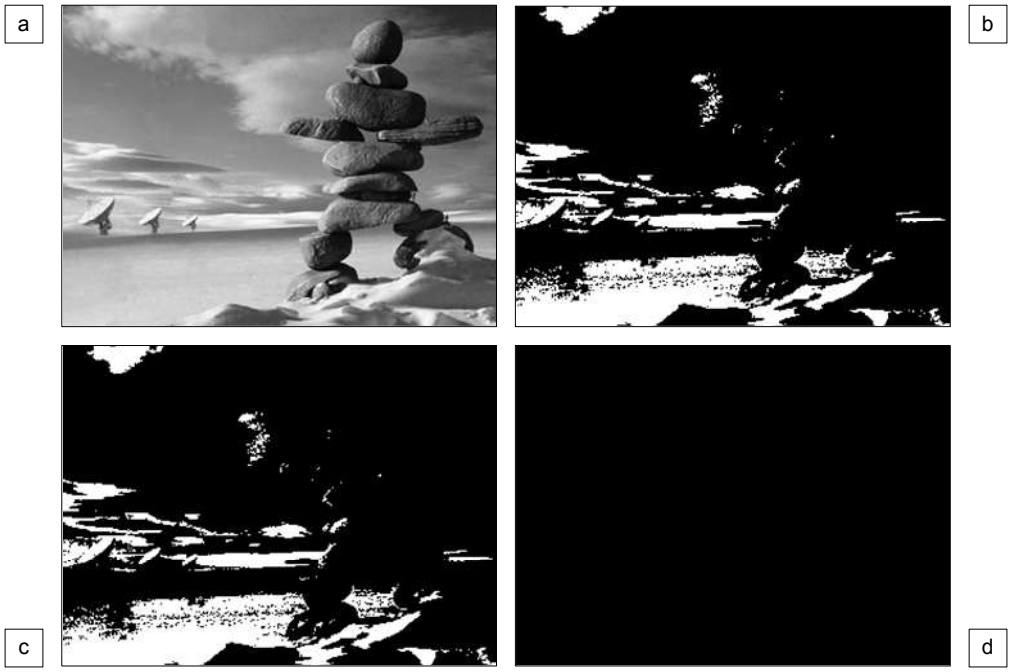

Figure 7.7: threshold results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

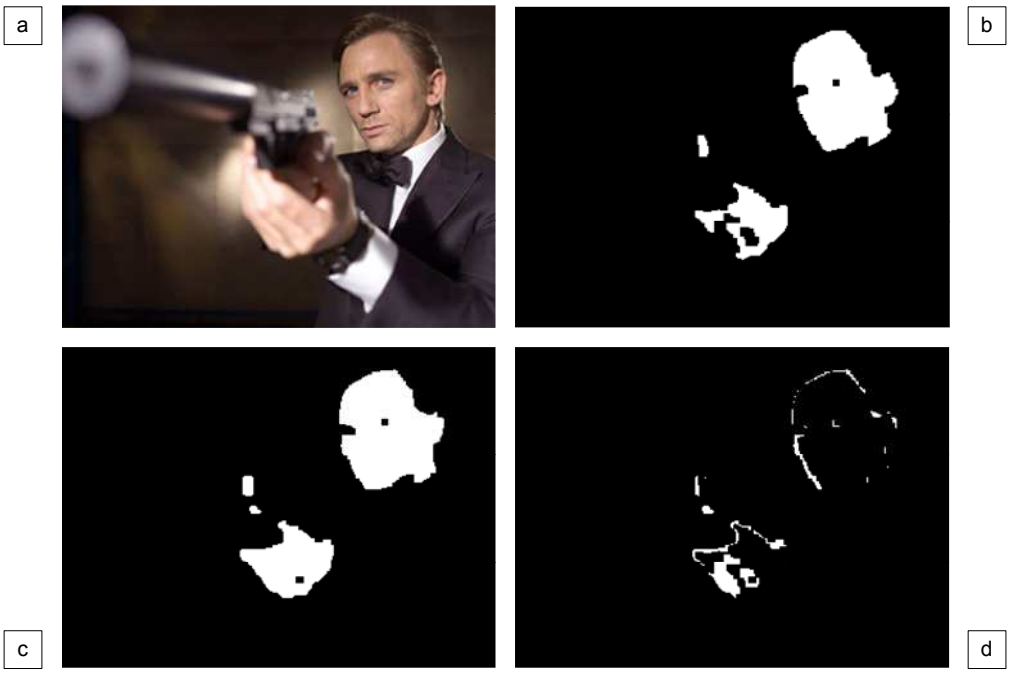

Figure 7.8: pixel processing pipeline skin segmentation results; original (a), software output (b), hardware simulation output (c) and difference between software and hardware (d)

uses 8 bits for the fractional remainder, the maximum error in the remainder is 2^{-8}. Increasing the amount of bits for the remainder would decrease the maximum error. However, the throughput will be lower since the computation of the remainder then requires more clock cycles.

7.2 Pixel Processing Pipeline

The implementation of the pixel processing pipeline comprises the components as discussed in section 6.2.1. After integrating these components, a manual stage by stage verification using behavioral simulation was performed. Then the simulation of processing a complete input image was performed. The verification results of this simulation are depicted in figure 7.8. The source image is shown in (a), the software generated output in (b) and the hardware simulation output in (c).

The difference comparison (d) shows that in certain cases the skin segmentation from the PPP differs in the output generated by the software. The differences can be attributed to the fact that the color space conversion and division components contain a fixed precision which introduce minor truncation errors in the conversion weights.

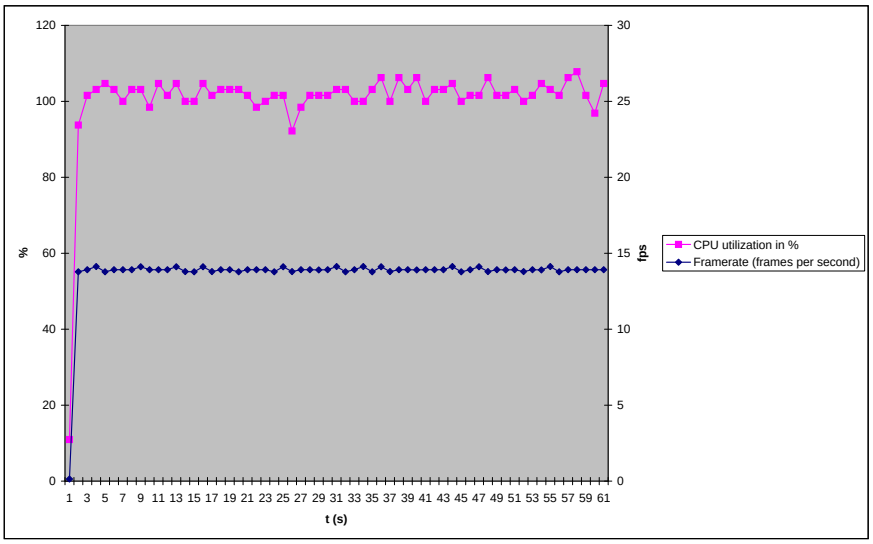

Figure 7.9: Software algorithm performance

7.3 Integral system performance evaluation

To perform integral system testing we used a stand alone workstation based on a 2,8 GHz Pentium 4 processor with 1024MB memory running Windows XP Pro as our reference and measurement system. The system setup additionally consists of a Philips ToUCam Pro webcam.

First the performance of the algorithm software implementation was estimated. Figure 7.9 shows both the CPU utilization and the achieved frame rate. As can be observed from the measurements, the software version is unable to achieve the real-time performance criterion of 25 frames per second. It achieves a frame rate close to 14 frames per second at a CPU utilization rate of 100%.

Figure 7.10 shows the results of the application demonstrator utilizing the hardware accelerator. The CPU utilization rate varies between 60% to 70% while the frame rate remains steady around 20 fps. This is a considerable improvement compared to the software version since the CPU utilization rate is significantly lower while yielding a higher frame rate. However, the real-time performance criterion of 25 frames per second is not met in the current implementation. Since the CPU utilization is not yet at 100%, the cause of the bottleneck remains in the network communication. However, the exact cause remains to be investigated.

The frame rate speedup gained with our hardware accelerated implementation is $20/14 \approx 1.43$. Moreover, the CPU utilization is 30% to 40% lower compared to the software implementation using the same computing system. Thus it allows for higher level applications such as hand gesture recognition to process the skin segmented output.

Future research needs to be performed to identify the exact network bottleneck of the current implementation which limits the achieved frame rate to 20 frames per second.

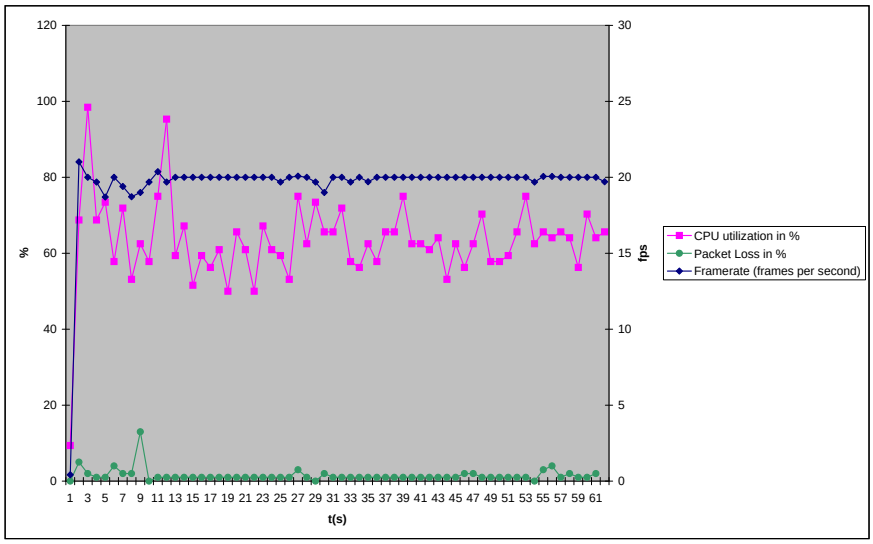

Figure 7.10: Hardware implementation performance

7.4 Conclusion

The evaluation as presented in this chapter shows the correct operation of the different components involved in the proposed accelerator system. Through behavioral simulation and testing of the implemented components and the pixel processing pipeline we verified that the skin segmentation algorithm has been successfully implemented in hardware. A demonstration system setup shows that the accelerator achieves a speedup of approximately 1.43 compared to its software implementation. Moreover, the CPU utilization rate was reduced from 100% to approximately 60% to 70%. Future work should concentrate on increasing the achieved frame rate and image sizes.

8 Conclusions and Future Work

This chapter concludes the discussion of this thesis project and provides suggestions for possible future work.

8.1 Conclusions

This project focused on the design and implementation of an FPGA accelerator for real-time skin segmentation. This report discussed the architectural exploration, design, verification and evaluation of both the accelerator architecture and its components.

We implemented the skin segmentation algorithm in hardware in order to meet the real-time requirements of 25 frames per second at an image resolution of at least 320x240 pixels. The demonstrator developed during this project achieves a frame rate of only 20 frames per second and a resolution of 320x240 pixels. The speedup gained with our hardware accelerated implementation is 1.43. Moreover, the CPU utilization is 30% to 40% lower compared to the software implementation using the same computing system. Thus it allows other software layers such as hand gesture recognition to more exactly interpret the skin segmented output.

Our design utilizes approximately 88% of the resources available in the XC2VP30 FPGA. Only 37% are required by the implementation of the pixel processing pipeline, the remaining resources are used by the accelerator architecture. The PowerPC processor operates at a clock frequency of 300 MHz while the pixel processing pipeline operates at a clock frequency of 100 MHz.

Besides the speedup mentioned above, our accelerator provides a portable solution due to the used standard network interface. This allows our accelerator to be used with a wide range of host PC's (both desktops and laptops).

8.2 Future work

Future work on this accelerator should concentrate on reducing the packet loss and the communication overhead. Furthermore, the design could also be implemented on a gigabit ethernet platform (such as presented in [13]) to achieve an input video resolution of 640x480 pixels at a frame rate of 25 frames per second.

It should be noted that the hardware demonstration application has not yet been optimized for execution speed. Several optimizations related to network protocol handling (as mentioned in amongst others [5], [10] and [16]) may be employed to reduce the communication overhead currently present within the demonstration application. Furthermore, research on possible network bottlenecks need be done to increase the frame rate of 20 frames per second to at least 25 frames per second.

A (simple) data compression algorithm can be employed to reduce the required network bandwidth. The overhead of such algorithm can be handled by the second PowerPC processor present in the FPGA. Furthermore, the following items specifically related to the pixel processing pipeline may be addressed in the future:

- Utilization of a higher clock frequency for the pixel processing pipeline. Currently, a clock frequency of 100 MHz is employed since this frequency is also used for the PLB bus. Higher clock frequencies might be possible but require specific design considerations regarding the interface with the PLB bus.
- The color space conversion component has been implemented with a shared multiply-accumulator unit to save FPGA resources. It is desirable to remove this sharing by instantiating three multiply-accumulator units to improve the throughput of the color space conversion component.
- Implementation of a "start of frame synchronization" for the local operations (uniform filter, local count, dilation and closing) present in the pipeline. When a pixel packet is lost during transmission, the entire image line will be lost and thus will not be processed by the pixel processing pipeline. The local operations however assume that all pixel packets arrive consecutively in order to determine when the image boundaries have been reached. When a packet is lost the internal counters will incorrectly determine when the image boundaries have been reached. Thus the local operations need to detect the start of a new frame to 'reset' their internal image boundary detection counters.

Bibliography

[1] S. Ahmad, *A usable real-time 3d hand tracker*, Proceedings of the Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers (1994), 1257–1261.

[2] A. Argyros and M. Lourakis, *Real-time tracking of multiple skin-colored objects with a possibly moving camera*, Lecture Notes in Computer Science **3023** (2004), 368–379.

[3] A. Benedetti and P. Perona, *Real-time 2-d feature detection on a reconfigurable computer*, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (1998).

[4] J.A. Boluda, F. Pardo, F. Blasco, and J. Pelechano, *A pipelined reconfigurable architecture for visual-based navigation*, Proceedings of the 25th EUROMICRO Conference (1999).

[5] J.S. Chase, A.J. Gallatin, and K.G Yocum, *End system optimizations for high-speed tcp*, IEEE Communications Magazine **39** (2001), 68–74.

[6] IBM Corporation, *Processor local bus architecture specification*, version 4.6 (2004).

[7] D. Crookes, K. Benkrid, A. Bouridane, K. Alotaibi, and A. Benkrid, *Design and implementation of a high level programming environmentfor fpga-based image processing*, IEE Proceedings Vision, Image and Signal Processing **147** (2000), 377–384.

[8] R. Duncan, *A survey of parallel computer architectures*, Computer **23** (1990), no. 2, 5–16.

[9] M. Flynn, *Some computer organizations and their effectiveness*, IEEE Transactions on Computing **C-21** (1972), 94.

[10] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier, *Tcp performance revisited*, Proceedings of the International Symposium on Performance Analysis of Systems and Software (2003).

[11] R. Gonzales and R. Woods, *Digital image processing*, second ed., Prentice Hall, 2002.

[12] A. Grama, *Introduction to parallel computing*, second ed., Addison Wesley, 2003.

[13] Xilinx System Engineering Group, *Gigabit system reference design*, Xilinx Application Note XAPP536 (2004).

[14] ———, *Xilinx university program virtex-ii pro development system*, Hardware Reference Manual (2005).

[15] J.L. Hennessy and D.A. Patterson, *Computer architecture - a quantitative approach*, third ed., Morgan Kaufmann Publishers, 2003.

[16] J. Huang and C. Chen, *On performance measurements of tcp/ip and its device driver*, Proceedings of the 17th Conference on Local Computer Networks (1992), 586–575.

[17] Intel, *Open source computer vision library.*

[18] R. Kjeldsen and J. Kender, *Finding skin in color images*, (1996), 312.

[19] M. Kolsch and M. Turk, *Fast 2d hand tracking with flocks of features and multi-cue integration*, **10** (2004), 158.

[20] P. Lam and S. Liew, *Udp-liter: An improved udp protocol for real-time multimedia applications over wireless links*, Proceedings of the 1st International Symposium on Wireless Communication Systems (2004), 314–318.

[21] L. Larzon, M. Degermark, S. Pink, L. Jonsson, and G. Fairhurst, *The lightweight user datagram protocol (udp-lite)*, Internet proposed standard RFC 3828 (2004).

[22] J.F. Lichtenauer, E.A. Hendriks, and M.J.T. Reijnders, *Robust skin color detection by elliptic cylinder fitting*, BMVC 2006 (submitted for publication).

[23] P. Mc Curry, F. Morgan, and L. Kilmartin, *Xilinx fpga implementation of a pixel processor for object detection applications*, Celoxica application note (2001).

[24] W. Najjar, B. Draper, A.P.W. Bhm, and R. Beveridge, *The cameron project: High-level programming of image processing applications on reconfigurable computing machines*, Proceedings of the Workshop on Reconfigurable Computing (PACT), Paris (1998).

[25] J. Postel, *User datagram protocol*, Internet standard RFC 768 (1980).

[26] ———, *Transmission control protocol*, Internet standard RFC 793 (1981).

[27] W. Stechele, L. Alvado Crcel, S. Herrmann, and J. Lidn Simn, *A coprocessor for accelerating visual information processing*, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (2005).

[28] Miguel A. Vega-Rodrguez, Juan M. Snchez-Prez, and Juan A. Gmez-Pulido, *Real time image processing with reconfigurable hardware*, Proceedings of the 8th IEEE International Conference on Electronics, Circuits and Systems, 2001. ICECS 2001. (2001), 213–216.

[29] Xilinx, *Virtex-ii pro and virtex-ii pro x platform fpgas: Complete data sheet*, version 4.5 (2005).

[30] Xilinx Corporation, 2100 Logic Drive,San Jose, CA, *Powerpc processor reference guide*, embedded development kit (edk) 6.1 ed., september 2003, version 1.1.

[31] H. Zimmermann, *Osi reference model–the iso model of architecture for open systems interconnection*, IEEE Transactions on Communications **28** (1980), 425–432.

Skin segmentation algorithm A

One of the first stages involved in the current gesture recognition system consists of identifying the image regions containing skin colors. This process is denoted "skin segmentation". The algorithm employed in this project is proposed in [22] and can described as depicted in figure A.1. The algorithm in this figure is annotated with the basic mathematical operations involved in each stage of the algorithm.

The algorithm produces an absolute difference output (for change detection) and a skin segmented output (for hand gesture recognition). In figure A.1, the left "branch" denotes the operations required to produce the absolute difference output, the middle and right "branches" denote the operations required to produce to skin segmented output. The following paragraphs discuss the change detection and skin segmentation parts of the algorithm. The numbers indicated correspond to the components in figure A.1.

A.1 Change detection

The algorithm provides an indication of change between two consecutive video frames. This is accomplished by calculating the absolute difference between the frames on a pixel by pixel basis. First, the input RGB pixel is converted to a grey scale pixel (1). Then the absolute difference between two corresponding pixels in consecutive frames is calculated (14). This can be accomplished by buffering the pixels in an image buffer (11). The resulting absolute difference image is smoothed using a uniform filter (16). A threshold (18) is applied to the smoothed image with which the sensitivity of the change detection can be regulated.

A.2 Skin segmentation

The skin model consists of two components, these are calculated using two color space conversions (2, 3). From each component a corresponding mean value is subtracted (4, 5). The results are then squared (7, 8) and added (9) to form a density compensated dissimilarity value. The grey value of the input pixel is also squared (6) before it divides (10) the density compensated dissimilarity. The result provides an indication of the skin likelihood and is thresholded using two different threshold values (12, 13). The second threshold (13) output is processed with a local count filter (15) followed by a threshold (17). This effectively filters the output of (13) of small regions. It is processed using a dilation (19) to thicken the contours of the resulting areas of (17). The logical AND function of the dilation (19) result and the first threshold (12) is then processed with a closing (20) operation to yield the skin segmented output.

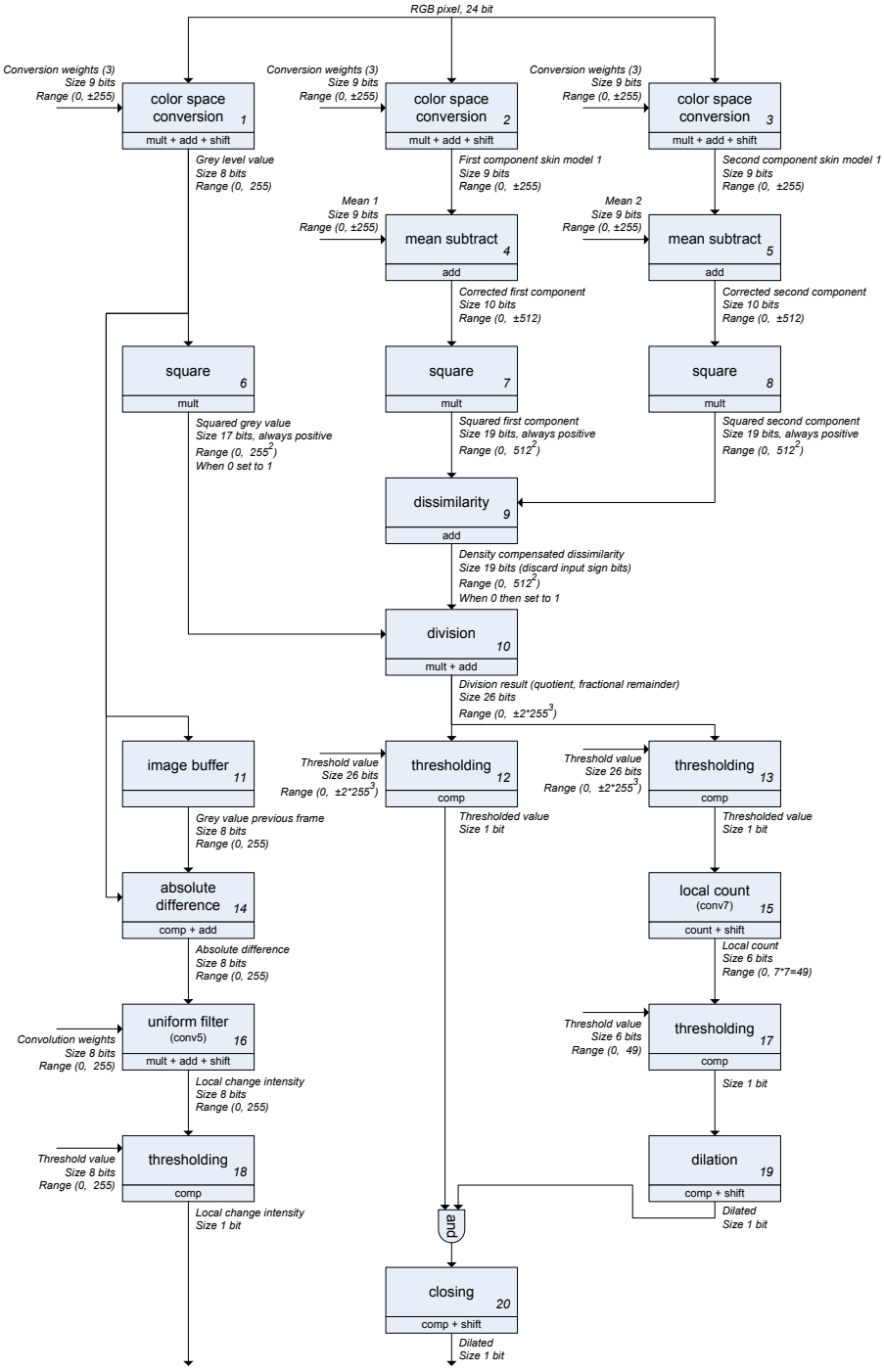

Figure A.1: skin segmentation algorithm

B PPP communication packets

This appendix provides a description of the pixel and configuration packets as utilized by the system. The first section discusses the ethernet header structure, then the pixel packet and configuration packet structures are discussed.

B.1 Ethernet packet header structure

The packet which is transmitted over the network connection comprises several protocol headers. Figure B.1 depicts the headers utilized in the current version of the system.

The first section of the header data consists of the ethernet header layer. It requires both a destination and source hardware addresses, (Media Access Control(MAC) address), to be present. Furthermore, at this layer it is also required to indicate which type of protocol is utilized. In all cases this field is set to Internet Protocol (IP).

The second section contains the IP layer header data and contains various IP control and status fields. Furthermore, it also contains the source and destination IP addresses and it indicates which protocol is utilized in the subsequent section. The current system utilizes the UDP protocol.

In the third section the UDP protocol headers are present. This header only contains the source and destination socket ports, the total length of the remaining data section and an optional UDP checksum. As mentioned in chapter 6, this checksum is permanently set to 0 in order to reduce the calculation overhead.

The subsequent section of the ethernet packet is determined by the type of packet being transmitted. This can either be a pixel packet or a configuration packet, both will be discussed in the next sections.

	byte number								Content:
offset	0	1	2	3	4	5	6	7	
0	Destination MAC address						Source ...		MAC
8	... MAC address				Protocol type (IP/ARP)		VER + IHL	TOS	MAC/IP
16	total datagram length		packet identification		flags + fragmentation		TTL	protocol=UDP	IP
24	header checksum		source IP address				destination ...		IP
32	... IP address		source port		destination port		length (header+data)		IP/UDP
40	checksum = 0		...	...	...	...	...	...	UDP

Figure B.1: common ethernet packet header structure

40	checksum = 0	reserved						IP/-
48	reserved							-
56	reserved							-
64	0xFF	type = 0x01	camera	reserved	0XFF	reserved	frame number	accelerator
72	0xFF	reserved	line number (y)		0xFF	reserved	column number (x)	accelerator
80	pixel data							pixel data
|								
240								

Figure B.2: pixel packet header structure

B.2 Pixel packet structure

Pixels are communicated using pixel packets as depicted in figure B.2. The indicated checksum field belongs to the UDP section of the packet.

The first 24 bytes are reserved in order to align the start of the pixel data at a 64 bit offset address. This requirement is posed by the PLB bus DMA protocol [6].

Then the PPP *packet type* is indicated, a pixel packet is indicate using type 1. The *camera ID* indicates which camera this pixel packet belongs to. The *frame number* contains the sequence number of the frame to which the pixels belongs to. The *line number* and *column number* indicate the x and y coordinates of the first pixel present within the pixel data section. Reserved bytes are also present within the packet to allow a modified implementation of the header in the future.

The pixel data can either contain input pixels (transferred from the host PC to the accelerator) or output pixels (transferred from the accelerator to the host PC). The input pixels are represented as depicted in the following table.

Grey T_{-1}	Blue	Green	Red

Table B.1: Input pixel format

where *Grey* T_{-1} represents the grey value of the corresponding pixel in the previous frame, *Blue*, *Green* and *Red* represent the blue, green and red color components of the input pixel.

The output pixels are represented as depicted in the following table.

0	Change	Grey T	Skin

Table B.2: Output pixel format

where *Change* represents the change estimation of the processed pixel, *Grey* T represents the grey value of the processed pixel and *Skin* represent the skin segmented output of the processed pixel. The empty (zero) field was added to ensure that the output pixel packets have the same size as the input pixel packets. This simplifies the software on both the PowerPC processor and the host PC.

40	checksum = 0	reserved							IP/-
48	reserved								-
56	reserved								-
64	0xFF	type = 0x02	reserved						accelerator
72	reserved		weight grey B		weight grey G		weight grey R		parameters
80	reserved		weight skin1aB		weight skin1aG		weight skin 1aR		parameters
88	reserved		weight skin1bB		weight skin1bG		weight skin1bR		parameters
96	uni w23	uni w22	uni w21	uni w15	uni w14	uni w13	uni w12	uni w11	parameters
104	uni w41	uni w35	uni w34	uni w33	uni w32	uni w31	uni w25	uni w24	parameters
112	uni w54	uni w53	uni w52	uni w51	uni w45	uni w44	uni w43	uni w42	parameters
120	reserved						uni shift	uni w55	parameters
128	division threshold 2 quotient (lccnt)			divt2 remd	division threshold 1 quotient			divt1 remd	parameters
136	uniform filter threshold				local count threshold				parameters
144	reserved								-
|									
240									

Figure B.3: configuration packet header structure

B.3 Configuration packet structure

The structure of a configuration packet is depicted in figure B.3. To indicate that the packet contains configuration data, the type field should be set to 2. The subsequent three section then contain the color space conversion weights. Currently the implementation allows for 9 bit signed conversion weights.

The uniform filter weights are presented using a single byte per weight. Each weight indicates either a multiplication factor of zero (0) or a shift left value (1-8) for the uniform filter kernel operation. The *uni shift* value indicates the amount of bits the result of the kernel operation should be shifted right.

Two division threshold values are present: the threshold value required before the local count (threshold 2 lccnt) operation and the threshold value which is directly connected to the AND port before the closing operation. Both threshold values consist of an integer quotient part and an 8-bit fractional remainder part. This remainder therefore provides an accuracy of 1/256th.

Both uniform threshold value and local count threshold value are present at the end of the configuration data packet.

C

PPP slave registers

The pixel processing pipeline contains 16 slave registers which are directly addressable by the processor. These registers are used as programmable parameters of the pipeline components an can be changed during run-time by sending configuration packets from the host PC to the system. Figure C.1 provides an overview of the currently employed slave registers and their respective function. The organization corresponds to the structure of the configuration packet structure as discussed in appendix B.

upper				lower			
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
			grey1c		grey1b		grey1a
			skin1a3		skin1a2		skin1a1
			skin1b3		skin1b2		skin1b1
conv5_w23	conv5_w22	conv5_w21	conv5_w15	conv5_w14	conv5_w13	conv5_w12	conv5_w11
conv5_w41	conv5_w35	conv5_w34	conv5_w33	conv5_w32	conv5_w31	conv5_w25	conv5_w24
conv5_w54	conv5_w53	conv5_w52	conv5_w51	conv5_w45	conv5_w44	conv5_w43	conv5_w42
						shamt	conv5_w55
					mean 2		mean 1
division threshold 2 (to local count)			div tr2_remd	division threshold 1			div tr1_remd
uniform filter threshold				local count threshold			

Figure C.1: PPP slave register organization

The following table provides an overview of the parameters, their function, range and default values.

Parameter name	Description	Range	Default value
grey1a	Grey scale conversion weight for the red color component	0 - +255	85
grey1b	Grey scale conversion weight for the green color component	0 - +255	85
grey1c	Grey scale conversion weight for the blue color component	0 - +255	85
skin1a1	Skin model 1a conversion weight for the red color component	-256 - +255	-61
skin1a2	Skin model 1a conversion weight for the green color component	-256 - +255	245
skin1a3	Skin model 1a conversion weight for the blue color component	-256 - +255	-185
skin1b1	Skin model 1b conversion weight for the red color component	-256 - +255	50
skin1b2	Skin model 1b conversion weight for the green color component	-256 - +255	-120
skin1b3	Skin model 1b conversion weight for the blue color component	-256 - +255	-162
conv5_w11	Uniform filter shifting weight 1,1 (top left)	0 - 256	1
conv5_w12	Uniform filter shifting weight 1,2	0 - 256	1
conv5_w13	Uniform filter shifting weight 1,3	0 - 256	1
conv5_w14	Uniform filter shifting weight 1,4	0 - 256	1
conv5_w15	Uniform filter shifting weight 1,5 (top right)	0 - 256	1
conv5_w21	Uniform filter shifting weight 2,1	0 - 256	1
	...		
conv5_w55	Uniform filter shifting weight 5,5 (bottom right)	1 - 256	1
shamt	Uniform filter result shift right amount	0 - 8	0
mean1	Mean subtract value for skin1a1	-256 - +255	-30
mean2	Mean subtract value for skin1a2	-256 - +255	-6
division threshold 1	Division quotient threshold to AND	0 - 65536	0
div tr1_remd	Division fractional remainder threshold to AND	0 - 255	4
division threshold 2	Division quotient threshold to local count	0 - 65536	0
div tr2_remd	Division fractional remainder threshold to local count	0 - 255	1
local count threshold	Threshold after local count	0 - 65536	15
uniform filter threshold	Threshold after uniform filter	0 - 65536	1250 (50*25)

Table C.1: PPP Parameter Descriptions

D Implementation data

This appendix describes the implementation results which provide insights into the resource utilization and timing of the individual pixel processing pipeline components. We implemented the accelerator architecture and its components using the Xilinx Platform Studio version 7.1i and the Xilinx Integrated Software Environment 7.1i. The synthesis tool used was Xilinx Synthesis Tool (XST) version 7.1.02i H.42.

The Virtex 2 Pro FPGA contains an array of interconnected Configurable Logic Blocks (CLB's). Each CLB contains 4 slices and 2 tri-state buffers. The slices are equivalent and contain amongst others 2 function generators and 2 storage elements. The function generators can be configured as either 4-input Lookup-Tables (LUTs), 16-bit shift registers or 16 bit Random Access Memory (RAM). The storage elements can be configured as D-type flip flops or level sensitive latches. The FPGA also contains Block RAM elements which can be used as dual port RAM.

The following tables present implementation details of the accelerator, pixel processing pipeline and the individual image processing components. The amount of finite state machines (FMSs), adders/subtractors and look-up tables indicated in these tables are results gained from HDL synthesis and do not necessarily present the low level synthesis results. The presented results assume an implementation with input image parameters of 320 width and 240 height. The resource utilization details for a 640x480 implementation are similar, the initial delay figures however differ. Resource utilization percentages indicate the utilization of the Virtex 2 Pro-30 [29] FPGA used in our implementation platform.

Accelerator system	
Resource utilization	
Slices	12138 (88%)
Flip Flops	11045 (40%)
4-input LUTs	17462 (63%)
Block RAM	124 (91%)
Multipliers	4 (2%)

Table D.1: implementation metrics for accelerator system

Table D.1 shows that the accelerator system uses 88% of the available slices and 40% of the available flip flops. Approximately 63% of the occupied slices are used as 4-input LUTs. Furthermore, the accelerator uses 91% of the available Block RAM, the majority of which can be attributed to both instruction and data memory required by the PowerPC processor. The current implementation requires 64 KB of instruction memory and 128 KB of data memory. Since a single Block RAM can contain a maximum of

2,25 KB [29], the amount of Block RAM resources used by the PowerPC memories is $(128 + 64)/2,25 \approx 86$.

Pixel Processing Pipeline	
Resource utilization	
Slices	5201 (37%)
Flip Flops	5751 (20%)
4-input LUTs	9096 (33%)
Block RAM	21 (15%)
HDL Synthesis	
FSMs	32
ROMs	111
Adders/Subtractors	29
Counters	15
Accumulators	2
Registers	2017
Latches	1
Comparators	16
Multiplexers	178
Logic shifters	2
Xors	96

Table D.2: implementation metrics for pixel processing pipeline

Table D.2 summarizes the resource utilization of the pixel processing pipeline. Approximately 37% of the available slices, 20% of the flip flops and 33% of the LUTs are used by the pixel processing pipeline. Furthermore, 15% of the available Block RAM is used.

The HDL synthesis results indicate the logic components involved in the pixel processing pipeline. It shows that 32 Finite State Machines (FSMs) have been inferred. These FSMs result from VHDL processes often performing a controlling function. The table shows that 111 ROMs have been synthesized, these are inferred from fixed lookup structures. Furthermore, 29 Adders/Subtractors, 15 Counters and 2 Accumulators have been inferred. Along with the 16 Comparators, these are often involved in supporting the FSMs.

Table D.3 shows the implementation and timing details of the local count component. The timing details show that this component requires 11 clock cycles to complete its operation. Consequently, the local count will produce an output 11 clock cycles after the start of a new pipeline cycle. However, this will not be the case during the first pipeline cycles. The initial delay indicates that initially 964 pipeline cycles with valid input to the component must occur before valid output is produced. This initial delay is typical for all components based on the neighborhood operation principle as explained in section 6.2.4. The reason is that these components must shift several image lines into their internal buffers before being able to calculate an output value.

The table also indicates that several Logicores are used in the design. These Intellectual Property (IP) cores are library components developed by Xilinx which can be

7x7 local count	
Timing	
Required clock cycles	11
Initial latency	964
Resource utilization	
Slices	415 (3%)
Flip Flops	215 (0%)
4-input LUTs	792 (2%)
BRAMs	1 (0%)
HDL Synthesis	
FSMs	1
ROMs	7
Adders/Subtracters	1
Counters	2
Registers	71
Multiplexers	1
Logicores	
Accumulator	
Equal comparator	
Greater Equal comparator	
FIFO	

Table D.3: timing and implementation metrics for 7x7 local count component

used as a black box component in the design. In this case, the component consists of an Accumulator (accumulating subsequent input values), Equal Comparator (compares two inputs and sets its output when these are equal), Greater Equal Comparator and First In First Out (FIFO) buffer.

For the following tables, the resource utilization, HDL synthesis results and the timing and Logicore details will not be explained further since these can be explained similarly to the discussions above.

Absolute difference	
Timing	
Required clock cycles	4
Initial latency	0
Resource utilization	
Slices	46 (0%)
Flip Flops	53 (0%)
4-input LUTs	65 (0%)
HDL Synthesis	
FSMs	1
Registers	10
Logicores	
Greater Equal comparator	
Adder/Subtractor	

Table D.4: timing and implementation metrics for absolute difference component

Binary closing	
Timing	
Required clock cycles	3
Initial latency	1286
Resource utilization	
Slices	334 (2%)
Flip Flops	260 (0%)
4-input LUTs	627 (2%)
HDL Synthesis	
FSMs	2
Adders/Subtracters	2
Counters	4
Registers	70
Multiplexers	2
Logicores	
none	
Components	
Erosion/Dilation (2x)	

Table D.5: timing and implementation metrics for binary closing component

5x5 uniform filter	
Timing	
Required clock cycles	9
Initial latency	645
Resource utilization	
Slices	1751 (12%)
Flip Flops	1156 (4%)
4-input LUTs	3253 (11%)
BRAMs	2 (1%)
HDL Synthesis	
FSMs	3
Adders/Subtracters	1
Counters	2
Registers	110
Multiplexers	2
Logicores	
Accumulator (2x)	
Equal comparator	
Greater Equal comparator	
FIFO	

Table D.6: timing and implementation metrics for 5x5 uniform filter component

color space conversion	
Timing	
Required clock cycles	21
Initial latency	0
Resource utilization	
Slices	109 (0%)
Flip Flops	117 (0%)
4-input LUTs	175 (0%)
HDL Synthesis	
FSMs	2
Registers	20
Multiplexers	2
Logicores	
Multiply accumulator	

Table D.7: timing and implementation metrics for color space conversion component

erosion/dilation	
Timing	
Required clock cycles	3
Initial latency	643
Resource utilization	
Slices	237 (1%)
Flip Flops	162 (0%)
4-input LUTs	444 (1%)
BRAMs	1 (0%)
HDL Synthesis	
FSMs	1
Adders/Subtracters	1
Counters	2
Registers	35
Multiplexers	1
Logicores	
Equal comparator	
Greater Equal comparator	
FIFO	

Table D.8: timing and implementation metrics for erosion/dilation component

24-bit signed adder	
Timing	
Required clock cycles	3
Initial latency	0
Resource utilization	
Slices	58 (0%)
Flip Flops	103 (0%)
4-input LUTs	32 (0%)
HDL Synthesis	
FSMs	1
Registers	8
Logicores	
Signed adder/subtracter	

Table D.9: timing and implementation metrics for 24-bit signed adder component

19-bit unsigned divider	
Timing	
Required clock cycles	11
Initial latency	5
Resource utilization	
Slices	571 (4%)
Flip Flops	674 (2%)
4-input LUTs	1009 (3%)
HDL Synthesis	
FSMs	1
Counters	1
Registers	15
Logicores	
Pipelined divider	
Greater Equal comparator	

Table D.10: timing and implementation metrics for 19-bit unsigned divider component

16-bit signed multiplier	
Timing	
Required clock cycles	4
Initial latency	0
Resource utilization	
Slices	78 (0%)
Flip Flops	136 (0%)
4-input LUTs	8 (0%)
Mult18x18s	1 (0%)
HDL Synthesis	
FSMs	1
Registers	11
Logicores	
Signed multiplier	

Table D.11: timing and implementation metrics for 16-bit signed multiplier component

32-bit pipeline register	
Timing	
Required clock cycles	2
Initial latency	0
Resource utilization	
Slices	39 (0%)
Flip Flops	68 (0%)
4-input LUTs	12 (0%)
HDL Synthesis	
FSMs	1
Registers	6
Logicores	
None	

Table D.12: timing and implementation metrics for 32-bit pipeline register component

32-bit threshold	
Timing	
Required clock cycles	3
Initial latency	0
Resource utilization	
Slices	56 (0%)
Flip Flops	69 (0%)
4-input LUTs	37 (0%)
HDL Synthesis	
FSMs	1
Registers	7
Logicores	
Greater Equal comparator	

Table D.13: timing and implementation metrics for 32-bit pipeline register component

1-bit delaying pipeline register	
Timing	
Required clock cycles	cycles reference component+1
Initial latency	latency reference component
Resource utilization	
Slices	40 (0%)
Flip Flops	40 (0%)
4-input LUTs	76 (0%)
BRAMs	1 (0%)
HDL Synthesis	
FSMs	1
Registers	8
Logicores	
None	

Table D.14: timing and implementation metrics for 1-bit delaying pipeline register component

16-bit delaying pipeline register	
Timing	
Required clock cycles	cycles reference component+1
Initial latency	latency reference component
Resource utilization	
Slices	48 (0%)
Flip Flops	55 (0%)
4-input LUTs	40 (0%)
BRAMs	1 (0%)
HDL Synthesis	
FSMs	1
Registers	8
Logicores	
None	

Table D.15: timing and implementation metrics for 16-bit delaying pipeline register component

E Design files overview

The project files are organized into the following categories: accelerator hardware and software files, individual pipeline components files, verification software files and demonstration software files. The organization of the files in these categories will be discussed in the following sections.

E.1 Accelerator project

The accelerator project files can be found in the *Accelerator* directory on the CDROM and are contained within a Xilinx Platform Studio project. The main project file is *system.xmp* and can be loaded into the software environment. Table E.1 lists the important directories and files of the accelerator project directory.

E.2 Pipeline components

Although the design files of the pixel processing components are contained within the accelerator project directory, these files are also available as individual projects. This was used for the design and simulation of the individual components. Table E.2 lists the directories for these components.

E.3 Verification software

The individual pixel processing pipeline components have been simulated and verified using comparison software. Different verification software has been developed for each component. The directory *Verification* contains these projects, each project is developed using Visual Studio C++ Express. The verification results can be found in the directory *Verification**verification_results*\ *.

E.4 Demonstration software

The demonstration software consists of two projects. The first project comprises a software equivalent version of the pixel processing pipeline (*PPP Software Demo*). Its performance is used as a reference to measure the speedup gained when utilizing the accelerator (*demo1*). The second project comprises a demonstration application using the accelerator. Both projects are implemented using Visual Studio C++ Express and are based on the Microsoft .NET platform. There was no specific reason to use this platform, other than that it is required by the Visual Studio Software to develop applications with a graphical user interface. The software equivalent project files can be found in the

Accelerator project files	
Main directory *Accelerator*\	
system.xmp	Xilinx Platform Studio Project file
system.log	System implementation flow log file
system.mhs	Hardware description file, states the components and their respective parameters and ports
system.mss	Software description file, states the operating system and drivers used
drivers\pixelpipeline_v1_00_a	Automatically generated test driver for the pixel processing pipeline, only provides test routines
*IMPACT PROJECTS\ **	IMPACT tool project files, used to program the Flash PROM with the accelerator configuration
pcores\pixelpipeline_v1_00_a\ data\pixelpipeline_v2_1_0.bbd	Black Box Definition file, contains the list of netlist files of the Logicores used in the accelerator
pcores\pixelpipeline_v1_00_a\ data\pixelpipeline_v2_1_0.pao	Peripheral Analysis Order file, contains the ordered list of VHDL files used in the project.
*pcores\pixelpipeline_v1_00_a\ devl\bfmsim\ **	Contains simulation files required for Bus Functional Simulation (BFM). Used to simulate the bus interaction of the pixel processing pipeline and the other bus connected components
*pcores\pixelpipeline_v1_00_a\ hdl\vhdl\ *.vhd*	Contains the VHDL files of the pixel processing pipeline and its components.
pcores\pixelpipeline_v1_00_a\ hdl\vhdl\PPP_ISE.ise	Xilinx ISE project file for development and simulation of the pixel processing pipeline
pcores\pixelpipeline_v1_00_a\ hdl\vhdl\user_logic.vhd	Top level design file of the pixel processing pipeline
*pcores\pixelpipeline_v1_00_a\ netlist\ **	Contains the logicores netlists
*PPP_Main\ **	Contains the software files for the PowerPC processor. The main software file is *main.c* and contains the initialization code. The additional files contain functionality for network protocol handling (IP, UDP, ARP, ETH and ICMP), interrupt initialization (int_init) and interrupt handling (handler_send, handler_receive).

Table E.1: overview of important accelerator project directories and files

Pipeline components files	
Main directory *Components*\	
absdiff\ *	Absolute difference component; project file, design files and simulation files
closing5\ *	Closing component; project file, design files and simulation files
conv5\ *	General 5x5 Convolution component; project file, design files and simulation files
csconv\ *	Color Space Conversion component; project file, design files and simulation files
erodil5\ *	Erosion/Dilation component; project file, design files and simulation files
localcount\ *	7x7 Local Count component; project file, design files and simulation files
padds24\ *	24-Bit Signed Adder component; project file, design files and simulation files
pdelay1\ *	1-Bit Delaying Pipeline Register component; project file, design files and simulation files
pdelay1_16\ *	1-Bit Delaying Pipeline Register component (FIFO depth of 16), design files and simulation files
pdelay16\ *	16-Bit Delaying Pipeline Register component; project file, design files and simulation files
divu19\ *	19-Bit Unsigned Divider component; project file, design files and simulation files
pmults16\ *	16-Bit Signed Multiplier component; project file, design files and simulation files
preg32\ *	32-Bit Pipeline Register component; project file, design files and simulation files
pthreshold32\ *	32-Bit Threshold component; project file, design files and simulation files
verification_results\ *	Verification results of the individual components

Table E.2: overview of important accelerator project directories and files

directory *Software**PPPSoftwareDemo*, the accelerator demonstration project files can be found in the directory *Software**demo1*.

E.5 Miscellaneous

The CDROM also contains miscellaneous files. The directory *Miscellaneous**TestImages* contains the original source images which have been used to verify the hardware components. The directory *Miscellaneous**makeECFmodel* contains the Matlab program for initializing the Accelerator parameters. It will generate an output file $c:\backslash pppparam.txt$ which contains the parameters for the pixel processing pipeline. These parameters will be read by the demonstration application *demo*1 and send to the accelerator upon start of the application.

Curriculum Vitae

Bart de Ruijsscher was born in Oostburg, the Netherlands on April 20, 1981. After receiving his HAVO diploma in 1999 he started the Electrical Engineering study at the Zeeland University of Professional Education. He gained industry experience at both Philips Remote Control Systems, Leuven and Siemens VDO Automotive (formerly known as Philips Car Systems), Eindhoven before receiving his B.Sc. in Electrical Engineering in July 2003. Shortly afterwards, he started the Master of Science in Computer Engineering study at the Electrical Engineering Department of Delft University of Technology. His research interests include hardware/software co-design and embedded systems for multimedia applications.