Circuit and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

2008

MSc THESIS

A SoC Solution for Fingerprint Minutiae Extraction

Michel van der Net

CAS-MS-2008-01

Faculty of Electrical Engineering, Mathematics and Computer Science

A SoC Solution for Fingerprint Minutiae Extraction

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Michel van der Net
born in Dordrecht, Netherlands

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

A SoC Solution for Fingerprint Minutiae Extraction

by Michel van der Net

Abstract

Fingerprint identification or verification is used more often in civilian applications. In the near future, Automatic Fingerprint Identification Systems (AFIS) can be found on mobile phones or smartcards. Most AFIS are however computationally intensive, and are designed for execution on large systems such as PC's. The main bottleneck is the minutiae extraction part. In this thesis a solution is presented to perform this part on a FPGA.

The solution consists of a modified version of the `MINDTCT` system, designed by NIST. It runs on an open-source micro-controller called `LEON2`, that is created by Gaisler. Both the software and hardware are modified to form a well operating minutiae extractor. The software is stripped and fixed-point conversion is performed. Also some algorithms are modified to enhance performance. For the `LEON` the right configuration needed to be found. The most time consuming parts of the software are accelerated by using co-processors. They where connected to the micro-controller, by using the CPI interface.

This project shows it is possible to run a fingerprint minutiae extractor on a FPGA, while using limited resources. Because the software is very computationally intensive, it takes an average of 60 seconds to complete one fingerprint. The accelerators will speed-up the system by almost 40%. The solution is based on the System-on-Chip (SoC) principle and therefore provides a perfect basis for a low-power fingerprint chip.

Laboratory	:	Circuit and Systems
Codenumber	:	CAS-MS-2008-01

To my parents and girlfriend for their endless love and support...

Contents

List of Figures

List of Tables

Acknowledgements

Doing my master on the TU Delft was a very educational experience. Although it was tough at times, I'm glad I could finish with this exciting project. I would like to thank the Circuits and Systems (CAS) group for assigning me this project. I would like to thank Dr. René van Leuken for his help and advice during this project.
During hard times, there where always three persons that supported me. Therefore I would like to express my great appreciation to my girlfriend and my parents.

Michel van der Net
Delft, The Netherlands
July 23, 2008

Introduction 1

Fingerprints are used in personal identification for over a century. They have several advantages over other biometric techniques, and are therefore commonly used in different identification and verification systems. Although fingerprints have a history in criminal identification, they are more frequently used in civilian applications as well. Recent examples of new types of Automatic Fingerprint Identification Systems (AFIS) are the ones used on embedded devices, such as mobile phones or smartcards.

Much research effort is going on to improve the quality of the systems. Reducing the execution times by software enhancements is also an important research topic. However with the new mobile types of AFIS in mind, research on how to efficiently execute algorithms in hardware is becoming more important. Most systems are executed on large systems with general-purpose processors and large memories, but this is not efficient enough for doing real-time identification on embedded systems. In this case the hardware resources are very limited and the software must be very efficient.

This thesis focuses on executing existing AFIS algorithms on a System-on-Chip (SoC) and therefore decreasing the processing time. The best available algorithms are picked and adapted to run on a small microcontroller. Then the most time consuming parts are mapped onto hardware accelerators in order to speed-up the execution.

1.1 Background

AFIS can be found more frequently in civilian applications, because of the increasing identity fraud. This creates a growing need for biometric identification. Fingerprints are often used, because they are the oldest and most mature form of biometric identification. This doesn't mean that it is completely reliable and that there are no drawbacks. Each biometric identification method has its own advantages and disadvantages. Fingerprints however show a good balance between the pros and cons [6].

The algorithms and computations differ for every AFIS, therefore some are more reliable than others. It is however important that every system gives good results, even for poor quality fingerprint images. In Figure 1.1 the basic block diagram of an AFIS is depicted. The input is always a gray-scale image, coming from a sensor. At the moment there are a few sensor types available, each with there own advantages and disadvantages [6]. First some pre-processing is done, this can be image enhancement to improve the quality of the image, image analysis to determine low quality areas in the image or a combination of both. After that the minutiae are extracted, which are the local ridge characteristics used in the matching step. Some post-processing is

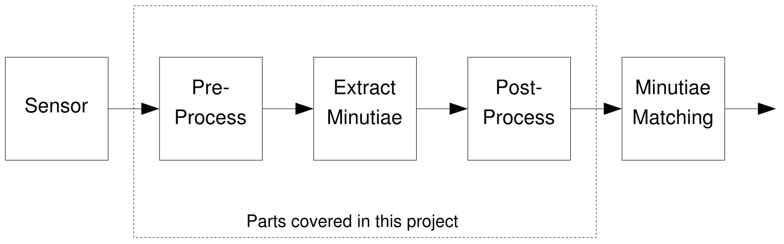


Figure 1.1: Basic block diagram of an AFIS

needed, it usually consists of removing false minutiae or assigning a quality factor to the extracted minutiae. The last step is minutiae matching, where the input is matched against fingerprints of a database. In this thesis only the pre- and post-processing and the extraction are targeted. Fingerprint matching requires fingerprint databases or a fingerprint sensor to live capture prints and test the system. Matching is left as future work, because a good quality usable matcher and a complete FPGA with sensor, were not readily available. Also minutiae extraction forms the bottleneck of the complete system.

The proposed system is based on the System-on-Chip (SoC) principle. Improvements on silicon process technology makes it possible to integrate a large number of basic blocks on a single IC. Such blocks can be processors, memories or other kind of hardware components. Normally each block would be a single chip and would be connected to each other on a circuit board. With SoC they are on a single chip, resulting in a small and fast low-power system.

1.2 Motivation

In the near future AFIS will be more frequently incorporated in mobile embedded devices. First the software needs to be modified, it must be made small and efficient enough. Doing the execution on large systems with general-purpose processors, is no longer satisfying enough. Doing the computations in real-time, calls for a small microcontroller with some hardware accelerators.

To extract minutiae from a fingerprint image, different algorithms are needed to get a reliable result. However which ones to take is still on debate. It is clear that some image enhancement is needed, because most of the fingerprint images are of poor quality. The system created in this project also performs quality identification of block or regions, which is to avoid extraction in low quality regions. Also there must be a minutiae extraction step to filter out the minutiae from the image. Because this puzzle has not been solved yet and the reliability of the extracted minutiae is still not high enough, the best readily available software system was chosen. The software is adapted and accelerated to give real-time results on a FPGA.

Another challenge is the creation of the system based on the SoC principle. This way the system can be put on a single chip with the advantages of lower power consumption, small size, smaller area and low production costs.

1.3 Thesis goals

In this thesis project six main goals can be derived, which are:

- Find a good performing usable minutiae extraction software system;
- Search for the best microcontroller for execution of the software system;
- Adapt the software so it has good performance and can run on the microcontroller;
- Profile and find the most time consuming functions or operations;
- Design hardware accelerators targeted at the microcontroller, that will speed-up the execution of the system;
- Create a synthesizable system of the microcontroller and its accelerators based on SoC principle, targeted at a specific FPGA board.

1.4 Thesis organization

This report is organized as follows:

- Chapter 2 first gives a background on fingerprint minutiae extraction. After that the chosen software system is discussed and motivated. Performance analysis results are presented.
- Chapter 3 discusses the modifications on the software system, needed to enable it to run on the LEON microcontroller. Also performance and profiling results are presented.
- Chapter 4 presents the background knowledge of the LEON processor, which will be used for executing the minutiae extraction system.
- Chapter 5 presents the architectures for the hardware accelerators, designed for the LEON to accelerate the most time consuming parts of the system.
- Chapter 6 shows the simulation and synthesis results for the implementation of the LEON and its accelerators on a FPGA board.
- Finally, Chapter 7 gives the conclusion and presents some directions for work to be done in the future.

Minutiae extraction software 2

This chapter starts by giving some background information of the fingerprint minutiae extraction process. Then the software system used, is presented and motivated. After that the system and all of its steps and algorithms are discussed in detail. Finally the performance of the system in terms of extraction results are presented and a motivation for the use of WSQ compression on the input images is given.

2.1 Background

Fingerprints are assumed to be unique for every person and remains unchanged over a life time. A fingerprint is formed by a pattern of ridges and valleys. Most often local ridge characteristics are used for identification, they are called minute details. Over 150 different details are identified, which are called minutiae [3]. Only two are used in most systems and are formed by ridge terminations (or endings) and ridge bifurcations [11]. For each minutiae some information must be stored. In most systems these are the x- and y-coordinates and the angle between the tangent to the ridge line at the minutiae position and the horizontal axis, see Figure 2.1. The systems used in this project also stores a quality parameter. If the minutiae is extracted from a low quality area in the image, a low value is assigned to it.

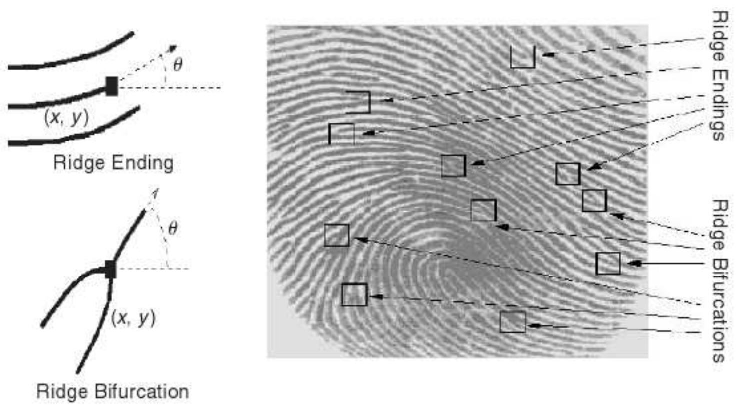


Figure 2.1: Minutiae types (left) and minutiae in a fingerprint image (right) [4]

Capturing minutiae is not an easy process, most often fingerprint images are of poor quality. Variations in skin condition are the main cause of poor quality. Scars, too wet, too dry fingers, dirt or bad contact with the sensor are examples. It results in false minutiae and the loss of true minutiae. Image enhancement tries to improve the quality of the fingerprint image, so that the minutiae extraction step gives more reliable results. Figure 2.2 shows a first level block diagram of the Minutiae extractor.

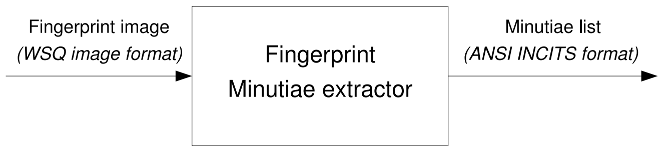


Figure 2.2: First level block diagram of the Minutiae extractor

Besides some small contrast enhancement the extractor performs image analysis. This is done to determine areas that are degraded and are likely to cause problems in the extraction step. The information collected in this step is used to assign a quality factor to each minutiae. After enhancement and analysis, different steps are sequentially executed resulting in a list of captured minutiae. A second level block diagram can be seen in Figure 2.3, more details on each of the steps can be found in Section 2.3.

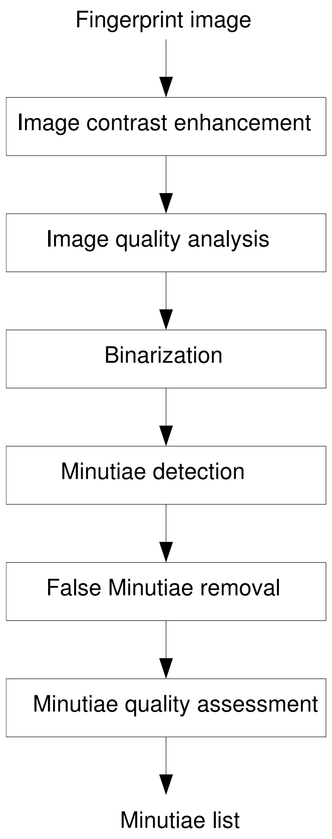


Figure 2.3: Second level block diagram of the Minutiae extractor

After enhancement and analysis, binarization is done. The gray-scale input image is transformed to a binary image, with only black or white pixels. Black pixels represents the ridges and white pixels the valleys. This process is needed, because the extraction algorithm can only operate on binary images. In most systems thinning is done after binarization, which reduces the ridges to one-pixel skeletons. Due to the implementation of the algorithm used, no thinning is needed.

After binarization, the actual minutiae detection can be performed. This step is straight-forward, the binary image is methodically scanned to identify pixel patterns that indicate endings or splittings. After the minutiae are extracted, some post-processing steps are done. The binarization step, greedy detection algorithm and poor image quality are three factors responsible for the introduction of false minutiae. Many different algorithms are executed to remove as much false minutiae as possible. In the end the system outputs the extracted minutiae.

2.2 Motivation

In this section the motivation for choosing the NIST MINDTCT software system is given. It is explained what algorithms are available and why the NIST system is the best option. Also it is explained why Matlab algorithms, which are used very often, are not the best option for this project.

2.2.1 Available algorithms

One of the most widely used fingerprint image enhancement algorithm is based on Gabor filtering and was first proposed by Hong et al [4]. It is based on normalizing the image, then do an orientation and frequency estimation of the ridges in the fingerprint and provide these to a Gabor filter. This method has been shown to give good results. It is however computationally expensive, because it involves a spatial convolution of the image.

Another method to enhance the image quality is proposed by Sherlock and is called directional Fourier filtering [10]. This method performs the convolution in the frequency domain and is therefore less computationally expensive. It only takes the orientation of the ridges into account and assumes the frequency to be constant. Therefore the method of Hong et al gives better results.

The third method is the STFT analysis method proposed by Sharat Chikkerur and is based on STFT analysis and contextual/non-stationary filtering in the Fourier domain [1]. The main advantage of this enhancement method is that it can instantaneously do frequency, orientation and region masks computation. After that a Fourier filtering is done.

The `MINDTCT` system developed by NIST for the FBI is a system that uses a different method for extraction [8]. No real image enhancement is done like in most systems that use the algorithms from above. Instead it performs an image analysis, where low quality areas in the input image are identified. It only performs a contrast enhancement when needed and doesn't improve the low quality areas. Instead it uses this information to assign a quality value to the detected minutiae. Image analysis in this system is also computational intensive, but far less than all of the algorithms above. Not using real image enhancement algorithms makes it possible to run this system on a microcontroller with limited hardware resources. The only drawback is the decrease of extraction quality.

2.2.2 C-code versus Matlab code

All algorithms discussed so far, except the `MINDTCT` system are only for fingerprint image enhancement and not for minutiae extraction. For the image enhancement algorithms the computation costs where already higher than that of the complete NIST system. Binarization, extraction and possible thinning still needs to be added to form a extractor. Also the algorithms are only available in Matlab code. It provides a good basis for developing algorithms, but this project requires C-code. It should be compiled for the LEON processor, which is based on C-code.

Two options are available, either transforming the M-code to C-code by hand or try to do it automatically by using the Matlab compiler. Option one will take months of work just for getting a software program. The complexity of the algorithms and the differences between coding in Matlab and in C is very high. The second option produces terrible looking code with an even worse performance. Also because the performance of the algorithms was already bad, a software system in C-code like the `MINDTCT` system is preferred. It seems to be the only well documented and fully usably minutiae extraction system that was readily available. It has good performance in case of execution time and only a slightly less performance in terms of minutiae extraction quality.

2.3 Description

In this section each of the steps, shown in Figure 2.3, are explained in detail.

2.3.1 Image Enhancement

In this step an algorithm is executed that improves the contrast of the image. First a histogram of the input image is evaluated. This is a plot of the pixels and there gray-scale values or intensities. If the pixel values are not well distributed over the histogram than enhancement is done. If they do, no enhancement is done.

The enhancement is done by using a technique called histogram equalization. First the histogram has to be build-up, this is done by analyzing all pixels and store all intensities in an incremental order. Then the mean intensity is determined by accumulating all intensities and divide it by the total amount of pixels. The highest and lowest tail intensities are searched. After this all pixels are compared to these values. If the intensity is lower than the lowest tail value it is set to this lowest value. The same is done for the highest value. The last step is the equalization. This is done by determining the equalization factor, which is computed by dividing the range (lowest tail value to highest tail value) from the highest intensity possible (255). For every pixel the lowest tail value is then subtracted for the pixel intensity and multiplied with the computed factor.

2.3.2 Image analysis

Because the image quality of fingerprints is often not high, it is critical to analyze areas in the image that are degraded and will likely cause problems in the detection step. Information about the quality can be found by measuring some characteristics. These are the directional flow of ridges, regions of low contrast, low ridge flow or areas with high curvature. Minutiae found in regions of low contrast, low ridge flow and high curvature are not reliable. Figure 2.4 shows the flow diagram of the image analysis step. In the following subsections the highlighted blocks will be explained in detail.

2.3.2.1 Low contrast analysis

In some areas it is impossible to detect minutiae, this is especially true for the background of the image and for smudges. No clearly defined ridges are present, and minutiae extraction should be avoided. The low contrast analysis algorithm generates a low contrast map, where low contrast areas are marked. It performs segmentation, which separates the foreground from the background and maps out all other low contrast areas in the image.

The analysis is done on image blocks. A block is marked low contrast if there is little dynamic range in the pixel intensities in that block. First the pixel intensity distribution is computed. By default the lowest and highest pixel intensities are ignored, so a much more stable portion of the distribution is used for the analysis. Then the distribution is compared with a threshold. If it is less, the block is marked low contrast. Basically the algorithm marks a block as low contrast if the center 80% of the pixel intensity distribution is not larger than 10 shades of gray on a 256 grayscale. In Figure 2.5 a typical low contrast map is shown, blocks with low contrast are marked with a white cross.

2.3.2.2 Direction analysis

Well-formed clearly visible ridges are essential to reliably detect minutiae. In this step a direction map is generated that represents areas of the image with sufficient ridge structure. It also records the general orientation of the ridges.
Orientation estimation is done for every block. By default the image is divided into 8x8 pixel blocks. To get a reliable estimation there must be enough local information, therefore each block is part of a window. The default size of a window is 24x24. The orientation is estimated for the block with all the information in the window, so the information of neighboring blocks is also used. This is called overlapping or smoothing. Each time the windows are shifted with 8 pixels. More information about

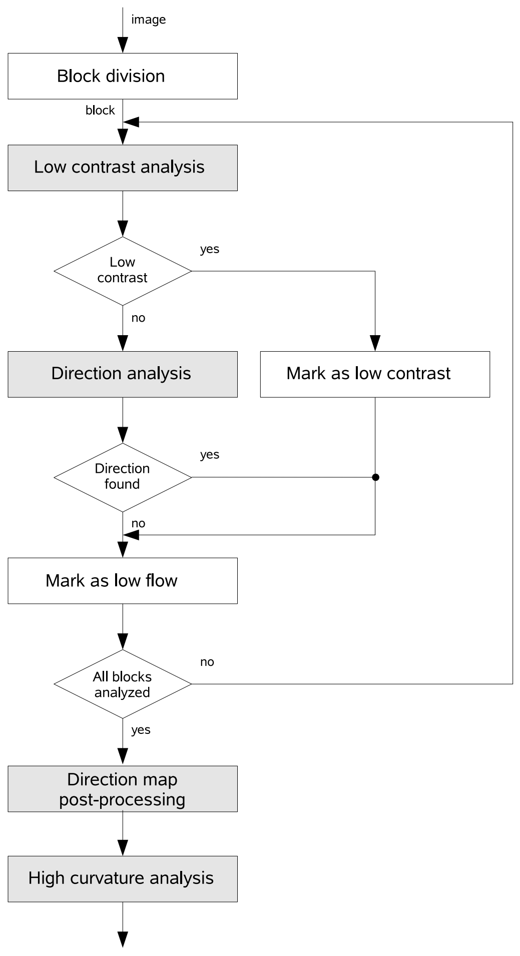


Figure 2.4: Flow diagram of the image analysis step

the implementation of blocks, windows and offsets can be found in [8] or in the source code.

For each block the window is rotated 16 times by default. Discrete Fourier Transform (DFT) is conducted at each rotation. For each rotation the pixels are accumulated along each of the 24 rows of the window, resulting in 24 rowsums. The amount of rotations is 16 so in the end there will be 16 vectors of rowsums. These vectors are then convolved with 4 waveforms of increased frequency. The first waveform represents ridges and valleys with a width of about 12 pixels. The second represents 6 pixels, the third 3 and the last one 1.5 pixels. Discrete values for the sine and cosine functions for each of the 16 vectors are computed by performing convolutions. They are multiplied with the rowsums, accumulated and squared. Adding the squared sine and cosine, produce a resonance coefficient that represents how well the vector fits one of the 4 waveforms.

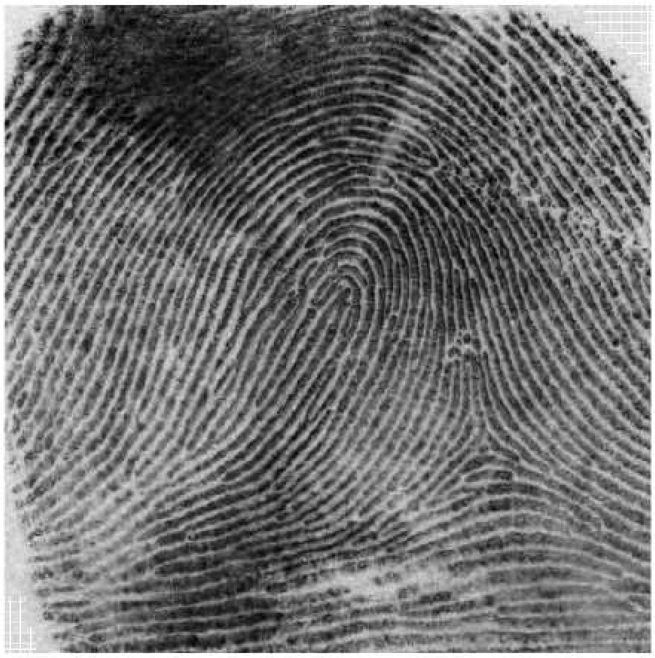


Figure 2.5: Typical low contrast map [8]

The dominant ridge flow direction for a block is the rotation with maximum waveform resonance. In Figure 2.6 a typical direction map is shown, the line segments indicating the orientation are centered in each block.

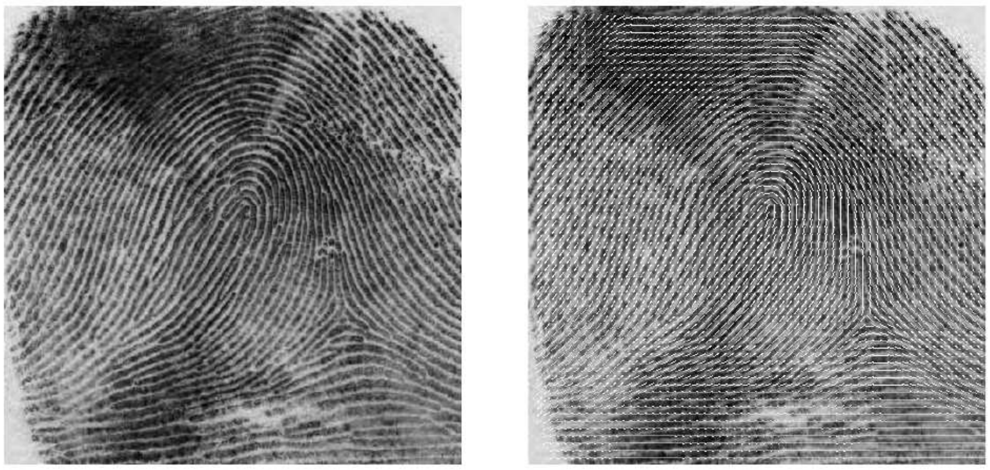


Figure 2.6: Typical fingerprint image (left) and its direction map (right) [8]

2.3.2.3 Direction map post-processing

In this step several algorithms are executed to improve the direction map that was initially generated in the direction analysis step. This step removes voids from the direction map by doing erosion and dilation. Directions from the map that are too weak or inconsistent are removed by using information from neighbors. It smooths the direction map, by computing the average direction of the neighbors. Neighbors with invalid directions are not used. The block is set to the average direction of its neighbors, if the number of valid neighbors is large enough and the average value is higher then a certain threshold. Also interpolation is performed, where the direction in a block is replaced by the weighted average of its neighbors, inversely proportional to the distance of the neighbors with the block.

2.3.2.4 High curvature analysis

Detecting minutiae in areas with high curvature is also problematic. Examples are the core and delta regions of a fingerprint. A core is defined as a center point of the semicircle pattern of a ridge shape. Delta as a center point of a triangle of a ridge pattern.

In this step a high curvature map is generated. Based on two different measures, the algorithm decides if a block is high curved or not. The first is vorticity, which is the amount of cumulative curvature incurred among the neighbors of each block. The second is curvature, which is the largest change in ridge flow direction between the analyzed block and its immediate neighbors. The algorithm details can be found in the source code. In Figure 2.7 a typical high curvature map is shown, regions with high curvature are marked with white crosses.

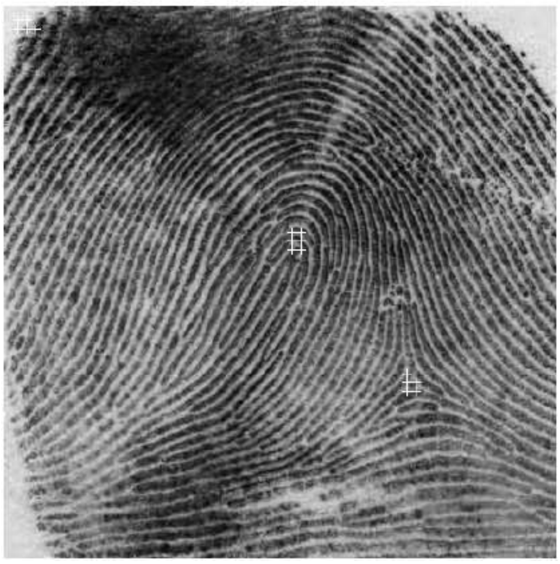


Figure 2.7: Typical high curvature map [8]

2.3.3 Binarization

Binarization is needed because the extraction algorithm needs a binary image as input. The algorithm uses the direction map, generated by the direction analysis step to determine if pixels in a block are white or black. If a block has no direction then all the pixels in that block are set to white. Otherwise a rotated grid is used to determine the value of the current pixel, based on the intensities of its surrounding pixels. With the pixel of interest in the center, the grid is rotated so that its rows are parallel to the ridge flow direction. Like in the direction analysis step, vector rowsums are formed by accumulating all pixel intensities of a row. By multiplying the center rowsum by the number of rows and comparing this value to the accumulated intensity of the entire grid, the value of the center pixel is determined. The center pixel is set to black, if the multiplied value is less than the threshold. Otherwise it is set to white. Figure 2.8 depicts the binarization results for a typical fingerprint image.

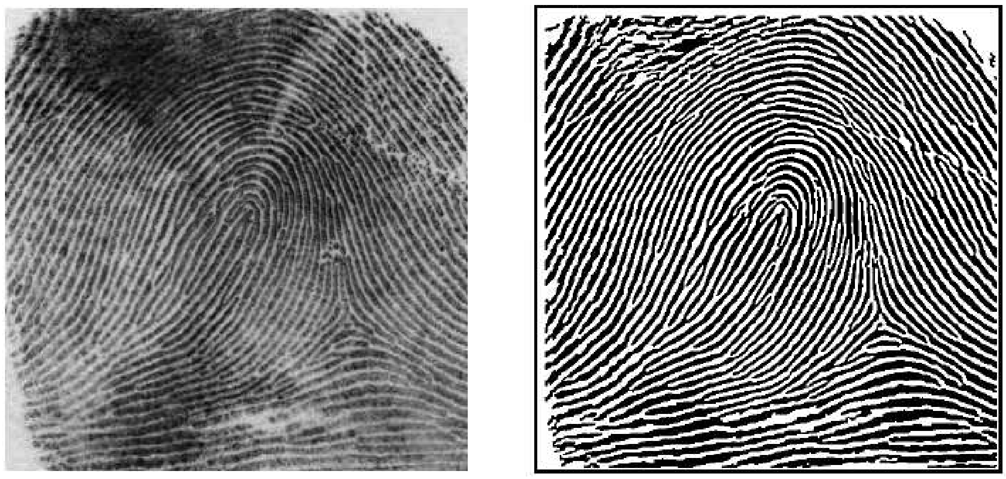


Figure 2.8: Typical fingerprint image (left) and its binarized version (right) [8]

2.3.4 Minutiae detection

In this step the minutiae are detected. Minutiae detection is a pattern matching problem, so a pattern matching algorithm is used. The algorithm performs a horizontal and vertical scan over the binary image, and tries to match pixel-pairs with 10 patterns that are stored in the system. With only 2 patterns, ridge endings can be identified. The other ones are used to identify bifurcations. A pattern can be appearing or disappearing, which depends on the direction the ridge or valley is protruding into the pattern. With a horizontal scan all vertically oriented minutiae are detected, with the vertical scan the horizontal ones. Figure 2.9 depicts the 10 patterns used by the algorithm. The star beside the patterns states that the middle pair of each pattern may be repeated one or more times.

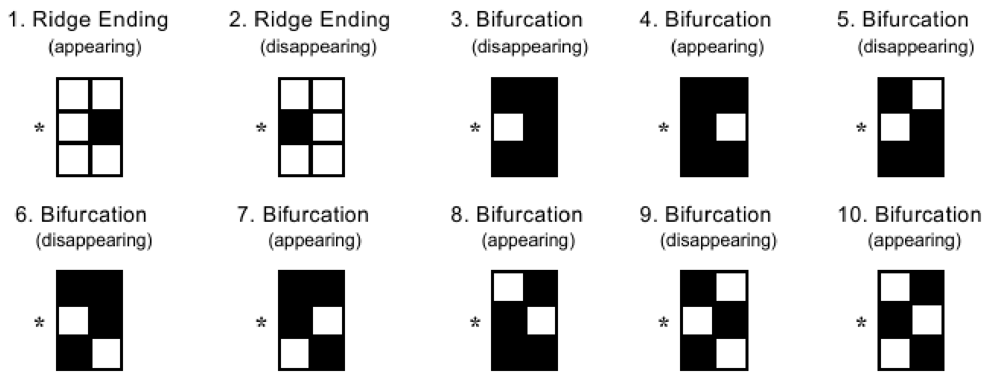


Figure 2.9: The 10 patterns used by the algorithm to detect minutiae [8]

2.3.5 False Minutiae removal

The minutiae detection algorithm is greedy, this means that the chance of missing true minutiae is small, but many false minutiae are introduced. This is mainly because the pattern matching is done with patterns of 6 pixels. False minutiae reduce the performance of the system, so much effort and code is dedicated to remove them. Different algorithms are executed to remove islands, lakes, holes, side minutiae, hooks, overlaps, low quality minutiae and too wide or too narrow minutiae. A description of all the algorithms is out of the scope of this report, more information can be found in [8] or in the source code.

2.3.6 Quality assessment

Despite of the many false minutiae removal steps, there still will be false ones in the final list. By assigning quality factors to minutiae it is possible to identify them. False minutiae will get a significantly lower quality factor than true minutiae. In this step a quality map is generated, that is build with the information of the low contrast, low flow and high curvature maps. In the quality map all blocks are assigned a quality factor ranging from high (5) to low (0). Minutiae quality is then based on two factors. The first one is the position of the minutiae in the quality map. If it is detected in a high quality area than the factor is increased. The second factor is based on intensity statistics. The mean and standard deviation within the immediate neighborhood of the minutiae are used to form a quality factor. By default the neighborhood is set to 11 pixels. If the mean is close to 127 and the standard deviation is larger than or equal to 64, a high quality factor will be assigned.

2.4 Performance

In this section the performance of the minutiae extractor system is discussed. First the quality of the extracted minutiae is discussed. A good quality system should extract *as many true minutiae as possible, while introducing as few false minutiae as possible. Because no sensor is used in this project, the fingerprint must be provided to the system in a different way. It is done by using a serial interface (UART). In order to reduce the communication overhead, the best image compression method had to be found. The effect of different types of image formats on the system is discussed. The original system has support for different formats, after a small study the best one is chosen.*

2.4.1 Minutiae extraction quality

Quality analysis on the minutiae extractor is not possible, unless the detected minutiae are compared with the results of a well known good performing system. Such a system is not available and therefore it is hard to determine whether the minutiae list is of good quality. However results for extraction and matching, are available for the NIST VTB system [7]. It uses the MINDTCT extractor as a basis. For the matching part the NIST BOZORTH98 matcher is used.

NIST spent 8 months on testing there VTB system and used databases ranging from 216 to over 600.000 fingerprints. The system has a false accept rate (FAR) of 1% with a true accept rate (TAR) of 99%. Also they performed a small comparison between the VTB system and some commercial products. The results were very satisfying and showed very equal performance.

2.4.2 Image input performance

No sensor is used in this project, so the fingerprint image will be provided to the system by sending the image from a PC to the hardware system. This is done by using an UART interface. To reduce the communication, the best compression method is chosen. The original system has support for 5 different image input formats, which are ANSI/NIST, WSQ, JPEGB, JPEGL and IHEAD. In order to reduce the system size, only support for one format was chosen. The ANSI/NIST and IHEAD formats are not widely accepted and are specific NIST image formats, therefore they where not chosen.

The target of the project is running a minutiae extractor on hardware with limited resources and memory, so therefore JPEGL is not an option. Although the format will give good results, due to the lossless compression, the image sizes are too large. That leaves the choice between the baseline JPEG format and Wavelet Scalar Quantization (WSQ). JPEGB has the advantages of wide acceptability. WSQ is developed by the FBI in order to reduce there database sizes, which became to large.

Many research papers state that WSQ is superior to JPEGB in terms of quality and image size on fingerprint images. A little performance research on the test database of 30 good quality fingerprints shows that indeed the WSQ format is far better than its JPEGB counterpart. In order to get good test results the original 30 fingerprints, which where in uncompressed PNG format, where converted to either JPEGB with 85% compression or in WSQ format. Then for both databases the minutiae where extracted and compared with each other. From Table 2.1 it can be seen that the file-size of WSQ is 72% smaller than JPEGB, while at the same time improving the extraction quality with almost 20%. The extraction quality is determined by the ratio between the amount of detected minutiae and minutiae with a quality factor equal or above 70.

Averages	**JPEG**	**WSQ**	**diff(%)**
filesize (KB)	42.6	11.9	-72.1
execution time (s)	0.36	0.39	8.3
detected minutiae	68	64	-5.9
high quality minutiae	28	39	39.3
overall quality	41.2	60.9	19.7

Table 2.1: Comparison between JPEG and WSQ

2.5 Summary

In this section the MINDTCT software system for the minutiae extraction is presented and motivated. The software is open-source and programmed in C-code. It contains many different algorithms, that together provides a full minutiae extraction system. Research shows it has a performance comparable with commercial products. Because no sensor is used in this project, a image compression method had to be found in order to reduce the communication overhead. This is because the PC sends the fingerprint image data, via a serial interface to the hardware system. The WSQ image compression format has the best performance on both size and quality, so this format was chosen as the input fingerprint format. The next section will present the modifications made on the software system.

Software customization 3

The software system described in the previous chapter, needs to be adapted for execution on a micro-controller. The system as it is developed by NIST, is targeted for execution on a PC. The source code and execution file are too large and contains too much overhead. Stripping of the source code is performed to remove all non essential code. The methods are explained in Section 3.1. The original system also contains many floating-point calculations. On CPU's like the Intel Pentium4 or Core2, floating-point computations can be done faster than integer versions. On micro-controllers, this is not the case. It is much faster to execute integer instructions. Furthermore it saves resources, because no expensive floating-point unit is needed. Therefore all floating-point computations are replaced by fixed-point versions. It is called fixed-point conversion and in Section 3.2 some details are presented.

3.1 Stripping

The original source code of the MINDTCT software system [8] is large, it contains much overhead. In order to let it run on a micro-controller, it must be stripped down to its bare minimum.

As explained in Section 2.4.2 the WSQ input format is used for the compression of fingerprint images. This means that all source code related to the other formats could be removed. It reduces the size of the executable by a significant amount.

The original system generates many different output files like: direction map, quality map, low contrast map, high curve map, etc. These files are not useful in this project. Only one output file is needed, the one that contains a list of minutiae in the ANSI/INCITS format. It can be used as an input for different minutiae matchers. By removing all code that is related to any of these other files, the execution file and source code is further reduced.

NIST improved many different functions in there system over time, but they left the original functions in the source code for performance comparison. Since these old functions are not executed and the new functions have better performance, these are all removed.

The original code contains many `fprintf` and `prinft` calls for generating log files and debug information. These are of no use in this project, and all calls are removed. This results in a significant performance increase.

To make sure that almost all overhead is removed and the program is stripped to its bare minimum, profiling is done to search for functions that will never be executed. Some functions will only be executed when, for example a fingerprint with low contrast or high curvature is processed. This makes the search more complicated. It is resolved by first performing dynamic analysis on fingerprints of the test database. Then a list of unused functions is generated and verified by performing static analysis on the source code. This way more then 10 extra functions could be removed.

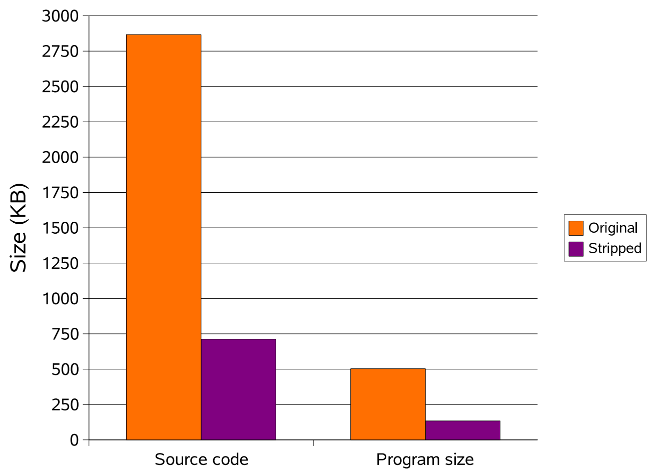


Figure 3.1: Comparison results between original program and stripped version

By removing different parts of the source code, the program was reduced by more then 73%. Figure 3.1 shows a comparison between the original program size and the stripped version. After the stripping the executable was small enough to run on a micro-controller, such as the LEON. After stripping, fixed-point conversion is the next step in transforming the system.

3.2 Fixed-Point conversion

In the following sections different parts of the conversion will be explained.

3.2.1 Tools and libraries

In order to reduce computation time and save hardware resources, the complete MINDTCT system is converted to a fixed-point version. This involves replacing all floating-point computations with fixed-point versions. This part of the project has shown to be very important and very time consuming. Tools and libraries for fixed-point conversion that are suitable for this project are not available. There are some compilers that can do the conversion automatically, but after some research it became clear that they are not efficient and sufficient enough. The same applies for the libraries that can be used to speed-up the conversion. There are some open-source libraries available, but all were not sufficient to be used in this project. They are either lacking many instructions, use very slow methods or are not developed in C-code.

3.2.2 Representation

A fixed-point number is a 32-bit integer number, with an imaginary point separating the integer part and the fractional part. With a floating-point number that point is not fixed and floats, with a fixed-point number this point is fixed and user defined. It can be signed or unsigned. For very large numbers one can decide to use 30 bits for the integer part, leaving 2 bits for the fractional part on unsigned numbers and 1 bit on signed numbers. This representation can represent large numbers, but has almost no fractional part. For the conversion, the right balance between the integer- and fractional part needs to be found for every floating-point variable that is used in the software. Also many integer values needs to be converted. For example when there is a operation on floats and integers.

3.2.3 Automatic conversion

In terms of this project, doing the conversion manually is a much better option than using a compiler. The main disadvantage is the amount of time it consumes. A large part of the time spent on this project, was devoted to the conversion. Automatic tools are not mature enough yet. All tools work in a similar way. First the programmer needs to annotate all the floating-point variables in the program. Then the compiler starts to do the conversion on each of those variables. The minimum and maximum values are determined and an accuracy is picked. Then the program is executed in a simulator and it is determined whether there are under- or overflows. Automatic tools can however not handle floating-point functions like: `sin`, `cos`, `atan2`, `sqrt`, etc. These functions still needs to be converted manually. Also picking the most accurate fixed-point representation is not sufficient enough in many cases. In terms of performance it is better to pick the least accurate, in cases where this is sufficient. Sometimes, even the most accurate fixed-point representation is not accurate enough. Then the only option is to modify the complete computation. An example is the scaling of very large numbers back to very accurate numbers. This is a worst-case scenario for fixed-point conversion, and is done many times in the software system. Manual conversion also results in more documented code, it is clear where modifications are made. This is handy for further development on the system, especially when trying to optimize the computations. It is easier to change the accuracy of functions or variables, to either improve performance or system output.

3.2.4 Manual conversion

The conversion is done from the bottom-up. This means the process is started with converting a leaf function. All the functions that call the leaf function are then converted. This is done until the top is reached. The process is repeated for the next leaf function. On all variables, dynamic and static analysis is performed to get the minimum and maximum values and the relation with other variables. The best fixed-point representation is then picked. After modifying a function, it is carefully tested and the program is executed to verify the output results. If the output differs too much from the original, the representation is modified. When this is no longer possible, when for example no more fractional bits are available, the computations are modified. Doing computations in a smarter way often results in having to use less accurate fixed-point numbers. Taking percentages is a good example, most often this is done by multiplying with a value between 0 and 1. This means that often a fixed-point number with a large integer part is multiplied with a number having a large fractional part. This way the result is very inaccurate. The computation in this case can be modified by a multiply and divide with larger numbers or even shifts. There are many more examples like: scaling, rounding, truncation, etc. Little less then a hundred functions are converted this way, resulting in an output that is very similar to the original one.

3.2.5 Fixed-point library

The second part of the conversion is the creation of a fixed-point math library. The main computations done in the system are: `add`, `sub`, `mul`, `div`. But also a lot of special functions are done like: `sin`, `cos`, `atan2`, `sqrt`, etc. All of these computations, except the `add` and `sub`, where added to a library in the form of macros. They can be used in the same way as normal functions, but with less overhead. For each fixed-point representation a slightly different macro is needed for each of the basic computations. The difference is mainly in the amount of shifts needed. Additions and subtractions on fixed-point variables can be done by using the integer versions. The only thing necessary is to convert the original floating-point or integer variables to a fixed-point variable with the right representation. For integer variables this can be done by shifting (left) the value, by the amount of fractional bits. For floating-point variables this can be done by multiplying them with $2^{fractionalbits}$. Multiplying two fixed-point numbers can be done by storing each of the operands in a 64-bit register and do an integer multiply. The result needs to be shifted to the right by the amount of fractional bits to get the right 32-bit value. Division can be done by storing the first operand in a 64-bit register, shifting it to the right by the amount of fractional bits and then perform an integer devision with the 32-bit operand.

3.2.6 Trigometric functions

The creation of fixed-point versions of the trigonometric functions takes more care and effort than the basic operations. The `sin`, `cos`, `atan2` needs to be estimated. The `CORDIC` method is chosen for doing this. It stands for `CO`ordinate `R`otation `DI`gital `C`omputer. It is a method for estimating trigonometric functions. Only `add`, `sub` and `shift` instructions are used, in combination with table lookups. This makes the method faster than other options. It is based on rotating the phase of a complex number. With each rotation it becomes closer to the desired value. The accuracy of the results is depending on the fixed-point representation used and the amount of iterations (rotations) in the `CORDIC` computation. The amount of rotations can easily be adapted and depends on the accuracy needed. In some parts the estimation can be rough, setting back these values results in an execution speed-up. However the estimation must be good enough, to keep the correct output of the system. In other parts the estimations can hardily differ from the floating-point versions and the most accurate fixed-point representation is needed. Then as much as 20 rotations are needed to get a similar output. For doing the `sqrt`, an integer function is first created. On a fixed-point number it can be done by first shifting the number by the amount of fractional bits and then call the integer `sqrt` function. For all other operations no special actions are needed and the integer versions can be used.

3.3 Performance

Two performance aspects of the fixed-point MINDTCT system are important. The first one is the comparison between the output results of the original MINDTCT system and the fixed-point version. This will be explained in Section 3.3.1. The second is the execution time of the new system compared with that of the original system. This will be explained in Section 3.3.2.

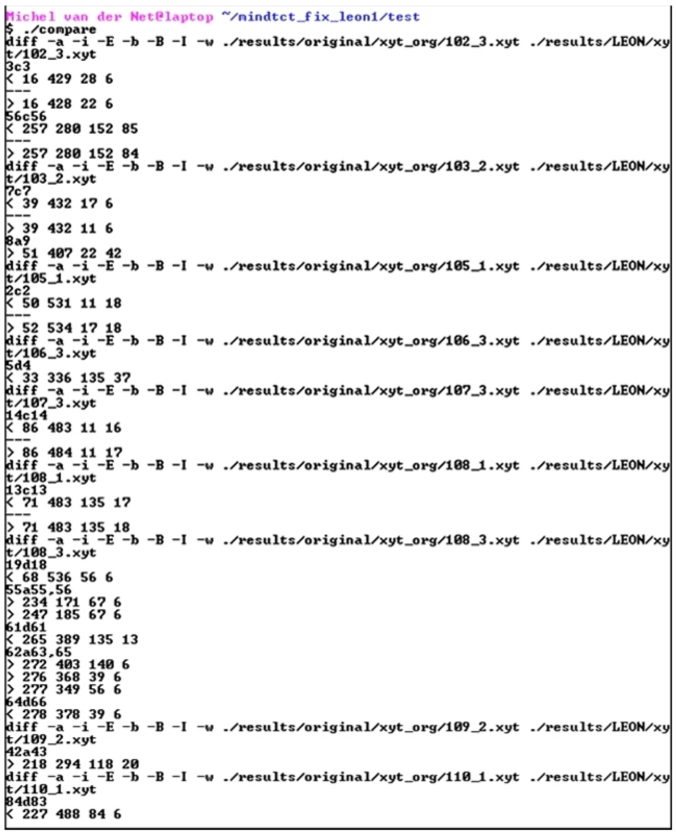
3.3.1 Minutiae extraction

It is important that as few as possible spurious minutiae are introduced, because of the conversion. Spurious minutiae with a low quality factor will be acceptable to some extend, because they will not be used in the matching process. However, having a high quality factor will not be acceptable. The amount of spurious minutiae should not be too large, because then the performance will decrease. The output file will become larger, and more processing needs to be done by the system. The most important factor is that the fixed-point system is able to extract all the high quality minutiae that are extracted by the original system. In Figure 3.2 a comparison between the output of the original and fixed-point version is shown.

Every minutiae with enough quality is detected by the new system. Only a couple of new low quality minutiae are introduced. Also some low quality minutiae are not detected. This does not decrease the quality of the system, because they are not used in the matching process. Moreover most spurious minutiae are coming from the WSQ decoding part, which is only being used in this project to decrease UART communication time. When a sensor is used the WSQ part becomes unnecessary and the quality of the output of both the original and fixed-point will improve. Especially the quality of the fixed-point version, because the decoder was very difficult to convert. The target for this project was to introduce as less differences between the original and new output as possible. In order to speed-up the execution, a more flexible fixed-point conversion can be performed. As long as the high quality minutiae are found, it is sufficient enough.

3.3.2 Execution time

After the fixed-point conversion, the performance could be estimated. The software will execute on the LEON micro-controller. It is important to do the performance estimate based on this architecture. A simulator that can simulate execution on the LEON is available and is called TSIM. It performs an instruction-level simulation. It is accurate and cycle-true, so it can provide good estimates of the execution times on the actual



```
Michel van der Net@laptop ~/mindtct_fix_leon1/test
$ ./compare
diff -a -i -E -b -B -I -w ./results/original/xyt_org/102_3.xyt ./results/LEON/xy
t/102_3.xyt
3c3
< 16 429 28 6
---
> 16 428 22 6
56c56
< 257 280 152 85
---
> 257 280 152 84
diff -a -i -E -b -B -I -w ./results/original/xyt_org/103_2.xyt ./results/LEON/xy
t/103_2.xyt
7c7
< 39 432 17 6
---
> 39 432 11 6
8a9
> 51 407 22 42
diff -a -i -E -b -B -I -w ./results/original/xyt_org/105_1.xyt ./results/LEON/xy
t/105_1.xyt
2c2
< 50 531 11 18
---
> 52 534 17 18
diff -a -i -E -b -B -I -w ./results/original/xyt_org/106_3.xyt ./results/LEON/xy
t/106_3.xyt
5d4
< 33 336 135 37
diff -a -i -E -b -B -I -w ./results/original/xyt_org/107_3.xyt ./results/LEON/xy
t/107_3.xyt
14c14
< 86 483 11 16
---
> 86 484 11 17
diff -a -i -E -b -B -I -w ./results/original/xyt_org/108_1.xyt ./results/LEON/xy
t/108_1.xyt
13c13
< 71 483 135 17
---
> 71 483 135 18
diff -a -i -E -b -B -I -w ./results/original/xyt_org/108_3.xyt ./results/LEON/xy
t/108_3.xyt
19d18
< 68 536 56 6
55a55,56
> 234 171 67 6
> 247 185 67 6
61d61
< 265 389 135 13
62a63,65
> 272 403 140 6
> 276 368 39 6
> 277 349 56 6
64d66
< 278 378 39 6
diff -a -i -E -b -B -I -w ./results/original/xyt_org/109_2.xyt ./results/LEON/xy
t/109_2.xyt
42a43
> 218 294 118 20
diff -a -i -E -b -B -I -w ./results/original/xyt_org/110_1.xyt ./results/LEON/xy
t/110_1.xyt
84d83
< 227 488 84 6
```

Figure 3.2: Output comparison between original program and fixed-point version

implementation. Only the evaluation version was available, therefore it was not possible to differ the cache or memory sizes. These were fixed to 4KB for both the instruction cache and data cache. There are much more restrictions, but it was possible to do a full execution of the MINDTCT system and perform performance estimates with a clock speed of 50 MHz.

Performance analysis is done on all 30 fingerprint images of the test database. Table 3.1 shows the averages for the 296x560 images of the database, 256x256 sensor sized images and uncompresses images. Future projects using this software system will have a sensor for capturing and inputing the fingerprint image. In that case no decompression method is necessary. From Table 3.1 it can be seen that this speeds-up the execution by an average of 25 %. Also the image size will be smaller, with a typical sensor size of 256 x 256 pixels. Therefore some performance estimates are done on fingerprint images with this smaller size. These values should give a close estimate of the performance of the FPGA implementation.

Averages	**30 database images**	**256 x 256 images**	**no WSQ**
execution time (s)	19,32	7,73	14,21

Table 3.1: Average execution times of the MINDTCT system

3.4 Profiling

With profiling the most time consuming parts of the system can be identified. A tool called `Gprof` can be used to profile the system on the PC. Although this gives good estimates of the most time-consuming parts, it is most likely that it will differ from the actual results when executed on the LEON. It has a different instruction set and a completely different architecture. The caches and main memory are much smaller and the pipeline is shorter. Therefore to do good profiling, it is performed with the TSIM simulator.

Function	**Ratio(%)**
`main()`	99.99
`get_minutiae()`	73.68
`lfs_detect_minutiae_V2()`	72.97
`gen_image_maps()`	27.26
`read_and_decode_grayscale_image()`	25.45
`read_and_decode_image()`	25.45
`wsq_decode_mem()`	25.45
`gen_initial_maps()`	23.70
`wsq_reconstruct()`	22.19
`join_lets()`	22.19
`dft_dir_powers()`	20.94
`binarize_V2()`	19.77
`detect_minutiae_V2()`	19.24
`sum_rot_block_rows()`	17.13
`binarize_image_V2()`	16.88
`dirbinarize()`	14.65
`match_1st_pair()`	10.58

Table 3.2: Ratio of the execution time, each function of the MINDTCT system takes

Table 3.2 shows the profile for the first fingerprint of the database. Only functions with an execution ratio higher then 10% are shown. The profile is almost the same for every fingerprint. Functions are shown in top-down order, starting with `main()`. Then `get_minutiae()` is the main function for extracting the minutiae, it calls many other functions which have a lower ratio. The WSQ decoding is part of `read_and_decode_grayscale_image()`. This means that this function and all of its child's are no candidates for the mapping on hardware. The child functions are: `read_and_decode_image()`, `wsq_decode_mem()`, `wsq_reconstruct()` and `join_lets()`. From the results it can be seen that the functions for the direction map generation are taking most of the time. Direction map generation involves a DFT analysis on all of the image blocks. This involves a lot of accumulations and multiplications. Also there is a lot of fetching from memory. The second part is the binarization of the image. The computations and methods involved are similar to the DFT analysis, except that a smaller grid is used and the input data is different. The only functions that are small enough and should result in an efficient mapping are: `dft_dir_powers()` and `dirbinarize()`. As can be seen from Table 3.2, the first has a ratio of almost 21% the second a ratio of 14.65%. Although the profile function order is almost the same for every image of the database the ratios are not, they will differ a little.

Function	**min (%)**	**max (%)**	**average (%)**
`dft_dir_powers()`	18.33	23.53	20.90
`dirbinarize()`	10.46	16.51	13.69

Table 3.3: Ratios of the execution time for the two mapping candidates

Table 3.3 shows the minimum, maximum and average ratios for both functions. It can be seen that, when adding the averages, the total ratio is 34,59%. A good estimate will then be a 50% speed-up over this number, which means 17,3%. More information can be found in Section 5. Also in Section 6 it is shown that the performance increase is much higher then the estimated 17,3%.

3.5 Algorithm modifications

Accelerators will be used to speed-up the DFT and binarization parts. Before these functions can be mapped onto hardware, some modifications are needed on the software. It consists of splitting the main functions into two parts, one where the fetching from memory is done and one for the computations. The result is one function that can be mapped onto hardware and one that remains to be done in software. In the original system the computations are scattered over several functions. For good mapping these computations are all collected into one function. This way there are two functions for the DFT part, which are: `load_block_from_mem()` and `sum_rot_block_rows2()`. The first function loads all needed pixels from memory and stores it into cache. The second function will be completely mapped onto hardware and will load the pixels from cache and do the computations. It should be noted that there is a small overhead by first loading from memory to cache and then from cache to the accelerator. One of the targets for this project was to keep the code clear and use as less assembly code as possible. Integrating the fetching and hardware calling, results in more complex assembly code.

The binarization part is also split-up into two functions, which are: `load_block_from_mem_binar()` and `dirbinarize_V2()`. The first one is implemented in the same way as `load_block_from_mem()` from the DFT part. The second function is modified in such a way that it can directly be mapped onto hardware. After the modifications the performance of the software is improved and the results can be seen in Table 3.4.

Performance analysis is done on the first three prints of the database and the ones with the highest and lowest execution times, which are fingerprint numbers 6 and 9. It can be seen that there is already a significant speed-up. This is mainly because of the better utilization of the bus, coming from the fact that first all pixels are loaded before doing the computations. In the original system only a small part of the pixels are loaded, which results in an alternating sequence of loads and computations. From the table it can be seen that not only the bus utilization is increased, but in some cases also the cache hit rate. This is also due to separation of loading and computation. The average speed-up achieved with the modifications is 2.83 s or 14.5%. Table 3.5 shows how much time the two DFT functions take.

	old time (s)	**ch & util**	**new time (s)**	**ch & util**	**speed-up (s) (%)**
Fingerprint1	19.55	98.3/85.7	16.96	98.3/100	2.59 (13.25)
Fingerprint2	20.87	97.2/95.4	17.64	98.5/100	3.23 (15.48)
Fingerprint3	19.18	97.2/92.4	16.32	98.5/99.8	2.86 (14.91)
Fingerprint6	21.27	97.2/94.8	18.02	98.4/100	3.25 (15.28)
Fingerprint9	16.43	98.4/80.4	14.20	98.6/93.1	2.23 (13.57)

Table 3.4: Performance results after the software modifications

DFT	**sum_rot_block_rows2()**	**load_block_from_mem()**
Fingerprint1	19.49 %	1.94 %
Fingerprint2	21.15 %	2.11 %
Fingerprint3	19.54 %	1.95 %
Fingerprint6	20.85 %	2.08 %
Fingerprint9	17.02 %	1.69 %

Table 3.5: Ratios of the execution time for the DFT functions

The ratio of the `sum_rot_block_rows2()` is also the maximum amount of speed-up that can be achieved with the DFT accelerator. Also the loading and overhead percentages are shown. Table 3.6 shows the same ratios for the binarization part.

When both DFT and binarization ratios are combined the maximum amount of speed-up with both accelerators can be found. Also about one half of the loading ratios is overhead, by adding assembly code and integrating this in the software an extra amount of speed-up can be achieved. Table 3.7 shows the results. An average speed-up of 29.68% can be achieved with an extra of 1.34% when the overhead is decreased.

Binarization	**dirbinarize_V2()**	**load_block_from_mem_binar()**
Fingerprint1	10.57 %	0.76 %
Fingerprint2	11.60 %	0.84 %
Fingerprint3	9.61 %	0.69 %
Fingerprint6	11.03 %	0.79 %
Fingerprint9	7.55 %	0.55 %

Table 3.6: Ratios of the execution time for the binarization functions

Totals	**maximum speed-up**	**extra speed-up**
Fingerprint1	30.06 %	1.35 %
Fingerprint2	32.75 %	1.48 %
Fingerprint3	29.15 %	1.32 %
Fingerprint6	31.88 %	1.44 %
Fingerprint9	24.57 %	1.12 %

Table 3.7: Maximum and extra speed-up results

3.6 Summary

Some modifications to the software where needed in order to provide execution on the hardware platform. It needed to be stripped down to its bare minimum, so all non essential code is removed. Also fixed-point conversion was needed. The conversion resulted in a system that is faster, needs less hardware and provides an output comparable with the original version. Profiling on the software shows the computational bottlenecks are in the DFT analysis and the binarization. Functions involved in both these steps are good candidates for execution with accelerators. The next section will present some details of the LEON micro-controller.

LEON 4

In this chapter the background knowledge of the LEON2 processor is presented. Executing MINDTCT on a desktop PC will not give any problems. The system is designed for it. The performance in terms of execution time is very good. The speed of modern processors and the amount of memory, are responsible for this. On a specific hardware system with limited resources, this is however very different. Therefore not only the software system needs modifications, but also the best hardware needs to be chosen.

In Section 4.1 some background information of LEON2 will be presented. Section 4.2 gives the motivation for choosing LEON2 instead of other architectures. Also an explanation is given why it is chosen instead of the newer and slightly faster LEON3. In Section 4.4 some knowledge about how to design and connect co-processors, is presented. Finally in Section 4.5 some VHDL implementation details are given.

4.1 Background

LEON2 [9] is a 32-bit processor core compliant with the IEEE-1754 SPARC V8 architecture [5]. It is open-source and completely available in synthesizable VHDL. The core is highly configurable, which means that it allows designers to optimize it for performance, power consumption, I/O throughput and area. Furthermore the architecture is based on System-on-Chip (SoC). It has a low complexity, it is highly configurable and has low power consumption. It is proven that LEON2 can be efficiently implemented on both FPGA and ASIC technologies, using standard synchronous RAM for both caches and register file. The basic processor core (pipeline, cache controllers and bus interface), consumes around 6000 LUTs and a clock frequency of up-to 50 MHz can be achieved on a FPGA. On a typical 0.25 micron ASIC, over 125 MHz can be reached while consuming less than 30.000 gates.

The micro-controller is implemented with separate instruction- and data caches (Harvard architecture), and has a 5-stage pipeline. The full SPARC V8 instruction set [5] is supported and a co-processor interface is available. It also has an unique debug interface, which allows non-intrusive hardware debugging and provides access to all registers and memory.

The architecture is centered around the AMBA Advanced High-speed Bus (AHB), to which the processor core and all the other high-bandwidth IP blocks are connected. Low-bandwidth blocks can be connected to the Advanced Peripheral Bus (APB), which is connected to the AHB bus via a bridge. The co-processor interface provides the highest bandwidth. Co-processor instructions are executed in parallel with the integer unit, if there are no data or resource dependencies.

4.2 Motivation

Choosing the right type of processor for this project, depends on certain aspects such as: good performance, low cost, support for C-code, configuration, FPGA support, SoC design and availability.

After some research it became clear that there are some commercial processor cores available, like Micro-Blaze (Xilinx), Pico-Blaze (Xilinx) and NIOS I&II (Altera). But these are all restricted to be used on expensive FPGAs of the brand that developed them. Although some of these cores have very good performance, an open-source core is preferable. In most cases they are: free, highly adaptable, not restricted to a certain brand and are readily available. Besides the LEON2, there are several other open-source cores available like: Open-RISC (OpenCores.org) and Open-SPARC (Sun).

For this project the LEON2 core is without a doubt the best open-source core. The Open-RISC core does not comply with the basic aspects needed by this project and the Open-SPARC is far too complex. Moreover research [2] shows that LEON2 is preferable in terms of performance and configurability. It confirms to all basic aspects and provides good documentation, support and software. It is readily available and can be downloaded from the Gaisler website [9]. It also comes with a co-processor (FPU) interface. This is important, because the performance of a hardware accelerator is much better when it is implemented as a co-processor instead of being connected to the bus.

At the time there is a newer and slightly faster and more optimized version of LEON available, named LEON3. It has a deeper pipeline and is better adapted to SoC designs. It however turns out that it has no co-processor interface. Modifications are needed to the integer pipeline and this is out of the scope of this project. There is a FPU interface, but it is specifically for the Gaisler FPU unit and is therefore not useable. LEON2 provides a clear interface to both the FPU and co-processor.

4.3 AMBA

The Advanced Micro-controller Bus Architecture (AMBA) defines an on-chip communications standard for designing high-performance embedded micro-controllers. It is chosen to be used in LEON2, because of its market dominance (ARM processors) and its good documentation. Three distinct buses are defined in the specification, which are:

- The Advanced High-performance Bus (AHB)
- The Advanced System Bus (ASB)
- The Advanced Peripheral Bus (APB)

Figure 4.1 depicts a functional diagram of LEON2 with its AMBA AHB and APB buses. The ASB system bus is not used in the processor. In Section 4.3.1 the AHB bus will be explained and in Section 4.3.2 the APB bus.

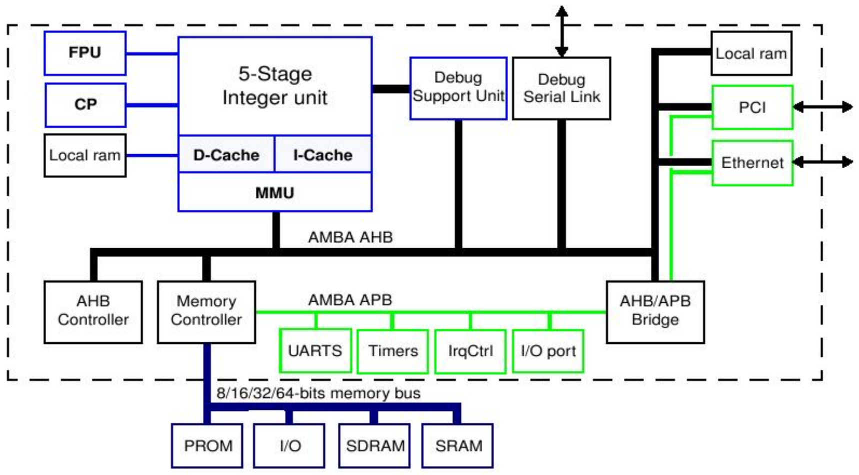


Figure 4.1: Functional diagram of LEON2 with AMBA AHB and APB buses [9]

4.3.1 AHB

The AHB bus has a high clock frequency and has high-performance. It is the performance bus to which all high-performing, high-bandwidth IP blocks communicate with the processor core. Such blocks are: integer unit, local ram unit, debug unit and the memory controller. Other user designed blocks, such as hardware accelerators can also use this bus.

The AHB bus is a multiple master, multiple slave bus. It is multiplexed (no tristate signals) and each master drives a group of signals to communicate with the slaves. Each master drives a set of signals grouped into a VHDL record called `HMSTO`. The output record of the current bus master is selected by the bus multiplexers and sent to the input record (`ahbsi`) of all slaves. The output record (`ahbso`) of the active slave is selected by the bus multiplexer and forwarded to all masters. A combined bus arbiter, address decoder and multiplexer controls which master and slave are currently selected. In Figure 4.2 the interconnection view is depicted.

4.3.2 APB

The APB bus is a single-master bus, suitable to interconnect units requiring only low data rates. It is interfaced with the AHB bus by a bridge. It is the only master on the APB bus. More than one bus can be connected to the AHB bus, by using multiple bridges.

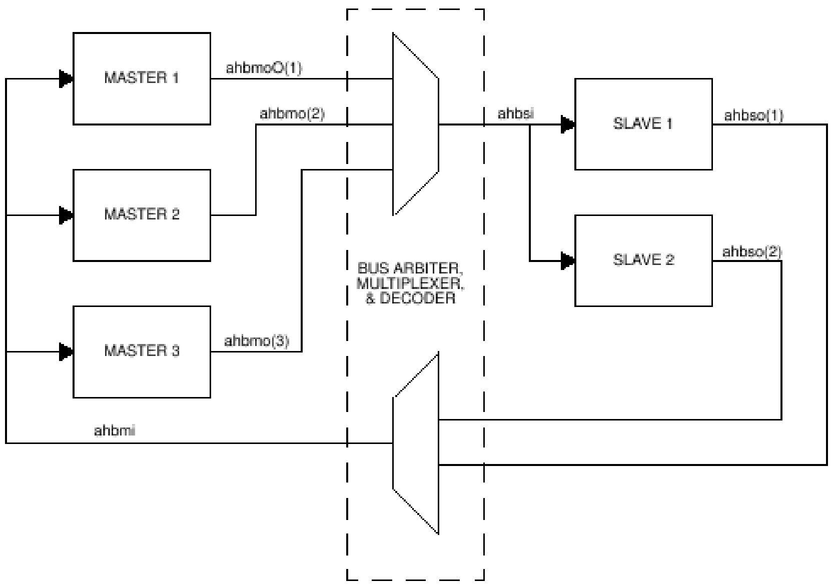


Figure 4.2: AHB interconnection view [9]

The APB bus is also multiplexed. The access to the AHB slave input (`AHBI`) is decoded and an access is made on the APB bus. The master drives a set of signals grouped into a VHDL record, called `APBI`. It is sent to all slaves. The combined address decoder and bus multiplexer controls which slave is currently selected. The output record (`APBO`) of the active slave is selected by the bus multiplexer and forwarded to AHB slave output (`AHBO`). In Figure 4.3 the interconnection view is depicted.

4.4 Co-processor

LEON can be configured to provide a generic interface to a special-purpose co-processor. The interface allows an execution unit to operate in parallel with the integer unit. One co-processor instruction can be started each cycle, but only if there are no data dependencies. When finished, the result is written back to the co-processor register file. A co-processor model for LEON2 is divided in two parts, the execution unit and core unit. The execution unit is the interface with the processor core and is provided with LEON2, it takes care of the parallel execution with the integer unit. In Figure 4.4 the interface is shown. The core unit is the actual implementation of the co-processor. To enable the co-processor, extra assembly instructions needs to be introduced. The FPU and co-processor interfaces are identical. The normal way of connecting the hardware accelerators, is by using the co-processor interface and creating special instructions. This involves some extra programming or even some compiler changes. When the FPU is not used another option is available, floating-point instructions like: FADDD, FMULD, FSQRT, FSUBD can be used by giving another operation meaning to them. Then in-line

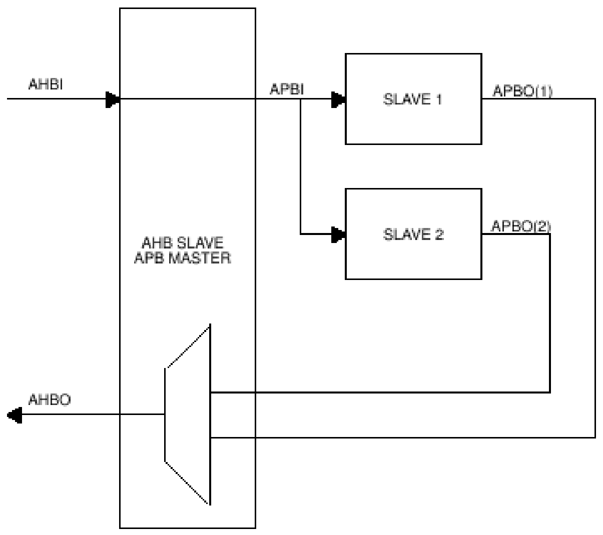


Figure 4.3: AHP interconnection view [9]

or normal assembly code can be used to operate on the accelerators by just using the FPU instructions. More detailed information on this can be found in Section 5.

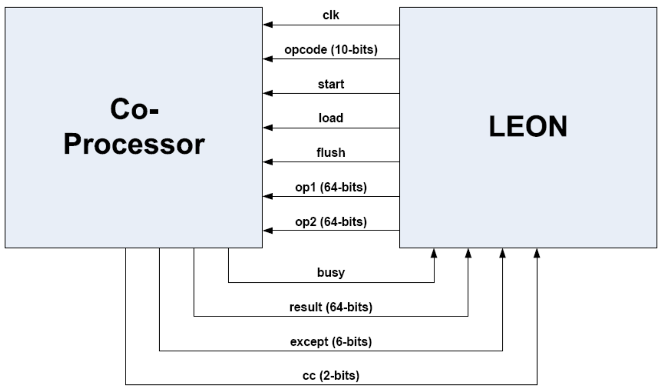


Figure 4.4: The co-processor and FPU interface

4.5 VHDL architecture

When connecting a co-processor to the LEON2 via the FPU interface, several files are involved. The LEON2 is completely written in VHDL, and is very modular. Therefore the complete implementation contains many VHDL files. Only three files are important when connecting a FPU or co-processor. First `device.vhd`, contains the configuration information. It is automatically generated when using the `xconfig` tool. With this tool the LEON can be set with a graphical interface. Here the FPU interface can be enabled, however there is no option for using it parallel with the integer unit. This must be done by hand, by changing the `fpu_config` variable interface option from serial to parallel. Here also the amount of registers can be set, it is by default set to 32 registers. The second file that needs some modifications is `fp1eu.vhd`, which contains the implementation of the FPU and co-processor interface. Here the actual co-processor needs to be added as a VHDL component and port mapped with the right signals. This is straight forward and it establishes a connection between the LEON and the co-processor, like the one depicted in Figure 4.4. The third and last file involved is `proc.vhd`, which contains all the port maps for the actual LEON implementations. It gets its information from the constants of `device.vhd`. So the only thing necessary is to check if the co-processor component is actually port mapped.

4.6 Summary

In this chapter information about the LEON2 processor core is presented. It is chosen for this project, because it confirms to all requirements. Furthermore it has shown to have good performance and is open-source. With the modifications to the MINDTCT system, described in Section 3, the software system can run on the LEON2. The hardware accelerators can be designed and connected to the LEON2 by either using the co-processor interface, FPU interface or the AHB high speed bus. The FPU interface is the preferred one, because then FPU instructions can be used to operate on the accelerators. No compiler changes or extra programming is needed this way. The next chapter will present the design and implementation details of the accelerators.

Hardware Accelerators 5

In this section the hardware accelerators will be presented. Two accelerators are created to speed-up the MINDTCT program. Section 5.1 gives a description for each of them. In Section 5.2 the design and implementation details can be found. Each accelerator is written in VHDL and connected to the LEON processor by using the FPU interface, which is the fastest interface possible for connecting accelerators to LEON.

5.1 Description

This section provides some details about the most time consuming parts of the MINDTCT system. Section 5.1.1 presents some details of the DFT power process. It is part of the direction map generation, presented in Section 2.3.2.2. The second accelerator speeds-up the binarization process, presented in Section 2.3.3. Some more details are given in Section 5.1.2.

5.1.1 DFT Power accelerator

Profiling shows that `sum_rot_block_rows()` is taking most of the time, see Section 3.4. It is called many times by `dft_dir_powers()`, which is called for each block in the image with sufficient contrast. The second function in computing the DFT powers is `dft_power()`. It takes less time than `sum_rot_block_rows()`, but is closely related to it. It is included in the hardware accelerator to increase performance. This way there is less communication between the accelerator and the LEON. The rotation function computes 24 rowsums for each of the 16 directions, so 24 * 16 = 384 numbers (13-bits) are computed. This means 384 * 13 = 4992-bits needs to be send back to the LEON. When including `dft_power()` only 64 DFT powers (fixed-point numbers) are computed, so 64 * 32 = 2048-bits needs to be send. Also with little hardware (some shifters and adders) an extra 3 to 4% performance increase is possible.

5.1.2 Binarization accelerator

The binarization process also takes a considerable amount of time. Three functions are responsible for the computations. The first `binarize_V2()` is the main function, which in terms calls `binarize_image_V2()` and `dirbinarize()`. The only one that can be mapped to hardware is `dirbinarize()`, because the other ones need the complete fingerprint image. This would either take to much memory space on the FPGA or results in large communication overheads. The only function that operates on individual pixels is `dirbinarize()`. The performed computations are comparable to the ones of the DFT process. For each pixel a surrounding window is used to compute the binary value (black or white). It is based on the ridge flow direction, associated with the block each pixel is in. The surrounding window or rotated grid that is used, is set to a width of 7 pixels and a height of 9 pixels. The pixel of interest is in the center of this grid. Rotating the grid parallel to the local ridge flow direction, computed in the direction map generation step, allows the algorithm to compute the binary value of the pixel.

In order to reduce communication overhead and do as much computations in parallel as possible, the binarization algorithm was slightly modified. The original method computes the binary value of pixels row by row. Then it is determined to which block the pixel belongs. The direction for that block is got from the direction map. Then the rotated grid is loaded from memory and the computations are performed to determine the binary value. The process is repeated for all pixels. The new algorithm makes use of the fact that all pixels in a certain block have the same direction. Block by block iterations are therefore more efficient then row by row. For each block the direction and the rotated grid can be loaded, and all the pixels in that block (64 pixels) can be computed. This reduces the communication overhead. Doing the computation in parallel on hardware will speed-up the process even further.

5.2 Implementation

This section shows the implementation details of the accelerators. It shows the full design with block and flow diagrams. It is also shown how the FPU instructions are used, in order to correctly control the accelerators.

5.2.1 DFT Power accelerator

Like described in Section 2.3.2.2 the DFT power computations are done on 24x24 pixel blocks. The rotating grid principle is used to do computations on 16 different directions. It is not sufficient to send 24x24 pixels to the hardware. For direction 0, when there is no rotation, all pixels are available and the rowsums can be calculated. But for the first rotation some extra pixels are needed. For the second rotation even more extra pixels are needed and so on. The accelerator needs all pixels in its buffer, in order to avoid communication with the LEON between the computations. The easiest way to realize this, is to send a 34x34 pixel block from the LEON to the accelerator. All pixels are then locally available. In total 34x34 = 1156 pixels needs to be send. From Figure 4.4 it can be seen that 128-bits can be send with a single instruction. A pixel is 8-bits so 128/8 = 16 pixels can be send with 1 instruction. In total 1156/16 = 73 instruction calls are needed to load a complete block. The `FADDD` instruction is used for this. Normally this instruction is used to add two floating-point doubles, but now it is used to send 16 pixels to the accelerator.

After sending all the pixels to the accelerator the execution can be started with the `FMULD` instruction. It requires no operands, and will return the first two DFT powers. This instruction triggers the actual computation and must only be called if all pixels are send to the accelerator. Because it takes more then 1 cycle to complete the computations, the busy signal is set high and turns low 1 cycle before completion.

For each direction 4 DFT powers will be produced, so in total 64 powers needs to be send back to the LEON. These powers are fixed-point numbers with a precision of 17-bits. Like stated in Section 5.1.1 2048-bits needs to be send. The interface allows 64-bits to be send with one instruction. So 2048/64 = 32 instruction calls are needed. For this the `FSUBD` instruction is used. Normally it is used to subtract two floating-point doubles. Its new function is to get 2 DFT powers per instruction call from the accelerator.

At the top-level the co-processor needs to be able to listen to instruction calls. Therefore the design contains a control unit. In Figure 5.1 the top-level block diagram is depicted. The control unit contains a state machine that handles the 3 FPU instructions. It is connected to the DFT execution unit and a pixel block RAM. This RAM module is used to store the 34x34 pixel block. The DFT execution unit performs the DFT power computations. Figure 5.2 shows the flow diagram of the control unit state machine. It starts in the idle state, where it waits for a start signal from the LEON. When it is asserted, the control unit reads out the opcode. If there is a `FADDD` instruction, it goes to the receive state. It gets the operands when the LEON asserts the load signal. The operands are stored in the pixel block RAM. The RAM is designed to store 16 pixels in one cycle. After one cycle it returns to the idle state.

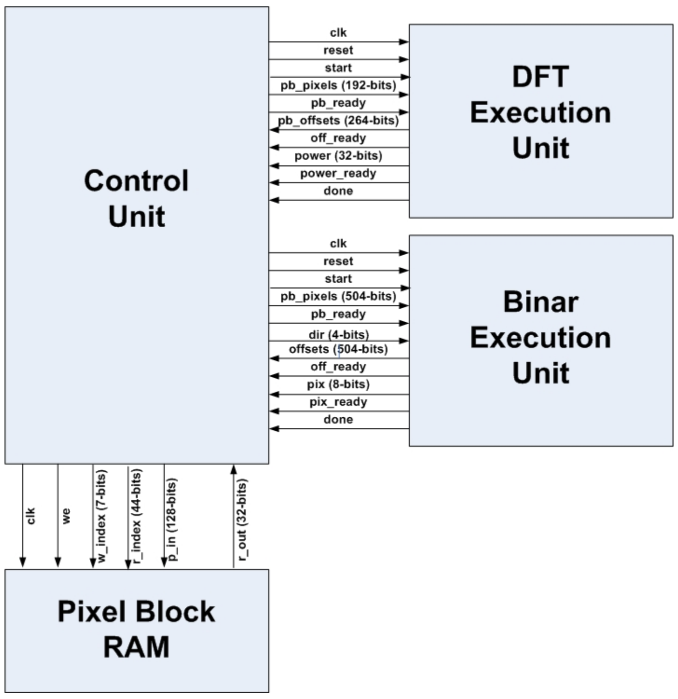


Figure 5.1: Top-level block diagram

If there is a `FMULD` instruction, the control unit goes to the execute state. It starts controlling the DFT execution unit, by asserting the start signal. Then it waits for the

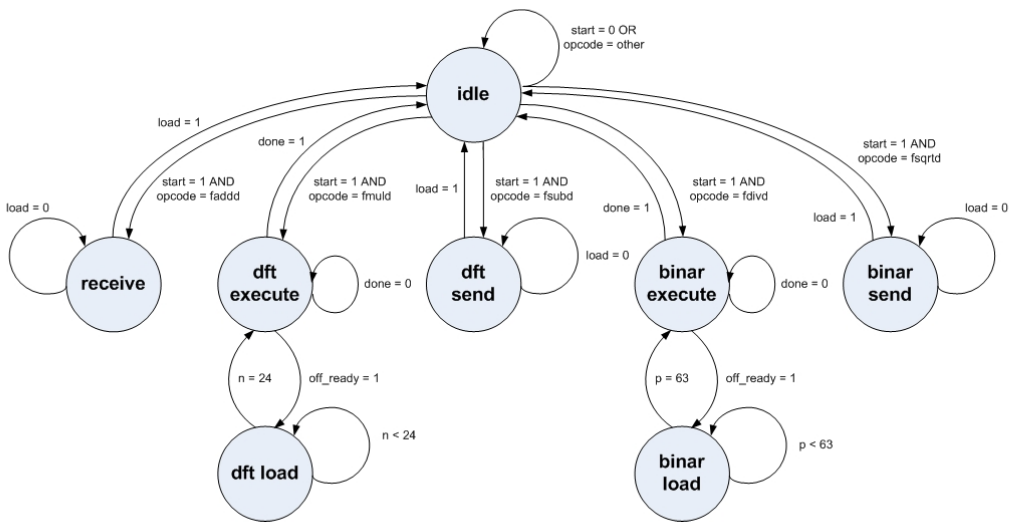


Figure 5.2: Flow diagram of the control unit

offset ready signal to go high. The DFT unit sends 24 grid-offsets to the control unit, which are used to get the right pixels from the RAM. The pixel block RAM is designed to provide 4 pixels in 1 cycle. Therefore the control unit goes into the load state until all 24 pixels are loaded from the RAM. As stated before, the RAM module can write 16 pixels in 1 cycle, but it can only read 4. The difference between the amount of data that can be read and written, is due to synthesis reasons. 16 pixels can be written, because it can be done with one address. The readout is different, 24 pixels are needed that are in different array positions in the RAM. Therefore 24 addresses are needed. To accomplish this, the synthesizer uses multiple block RAMs in parallel. Therefore only 4 addresses are used and 4 pixels can be readout in 1 cycle. It is done to reduce the overhead and still get multiple pixels per cycle out of the RAM. Instead of block RAM, distributed RAM can also be used, but then LUTs are wasted. When all pixels are loaded from the RAM, it asserts the pixels ready signal and goes back into the execute state. The DFT unit keeps asserting and receiving the offset ready and pixels ready signals, until the rowsums are computed. Then it starts to compute the DFT powers. Each time a power is computed the power ready signal is asserted and the power is send to the control unit. A buffer is used to hold all 64 powers. When the computations are finished, the done signal is asserted and the control unit goes back into the idle state.

The DFT execution unit also contains a control unit, connected with two ROMS. Figure 5.3 shows the block diagram. The grid-offsets ROM contains 9216 offset (11-bits). So the ROM has a size of 9216 x 11 = 101.376 bits. It can provide 24 offsets each cycle. The other ROM contains the sin and cos values needed for the DFT computations. It contains 96 sin and cos fixed-point values. The fixed-point values are normally 32-bits, but the values are between 1 and -1 and therefore only 17-bits are needed for fixed-point numbers with a precision of 17-bits. This ROM has a size of 192 x 17 = 3264 bits. Each cycle a sin and cos value can be provided.

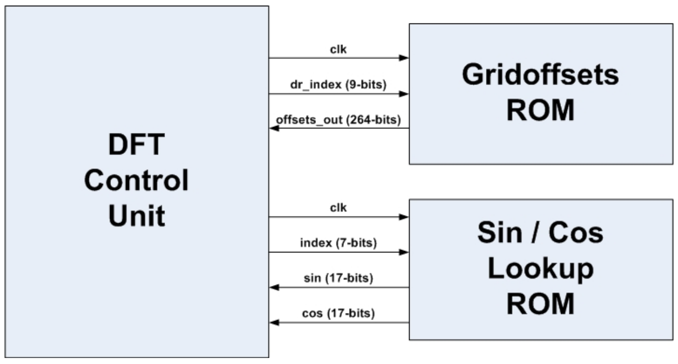


Figure 5.3: Block diagram of the DFT execution unit

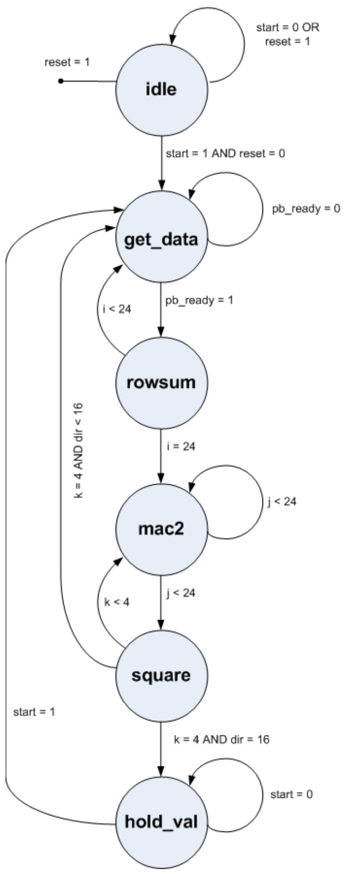


Figure 5.4: Flow diagram of the DFT control unit

Figure 5.4 shows the flow diagram of the DFT control unit state machine. It starts in the idle state, where it waits for the start signal to be asserted by the control unit. After this it goes to the get_data state. Here it gets 24 offsets from the grid ROM, asserts the offset ready signal and waits until the control unit has loaded the 24 pixels out of the RAM and asserts the pixels ready signal. When the pixels are available, it goes into the rowsum state and computes the rowsum.

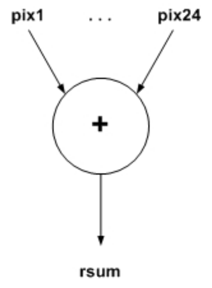


Figure 5.5: Rowsum computations

Figure 5.5 depicts the computations performed in the rowsum state. In one cycle 24 pixels (row) are accumulated and the result is shifted to form a 32-bits fixed-point number. It is stored in a buffer until all 24 rowsums are computed and the DFT execution unit switches to the mac2 state. In this state two multiply and accumulates are performed on each of the rowsums. Each rowsum is multiplied and accumulated with the corresponding sin and cos values. Figure 5.6 shows the computations performed in the mac2 state.

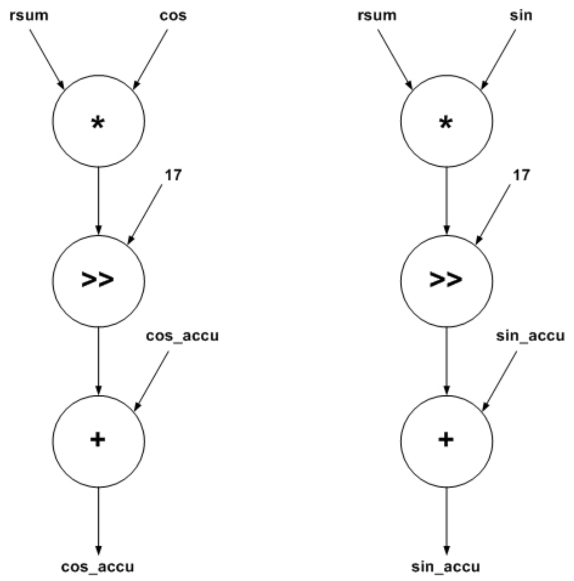


Figure 5.6: Mac2 computations

The square state is the last step in the process. It is depicted in Figure 5.7. Here the two accumulated values from the mac2 step are squared to form the DFT power. The DFT control unit repeats the process until all rotations are performed and all 64

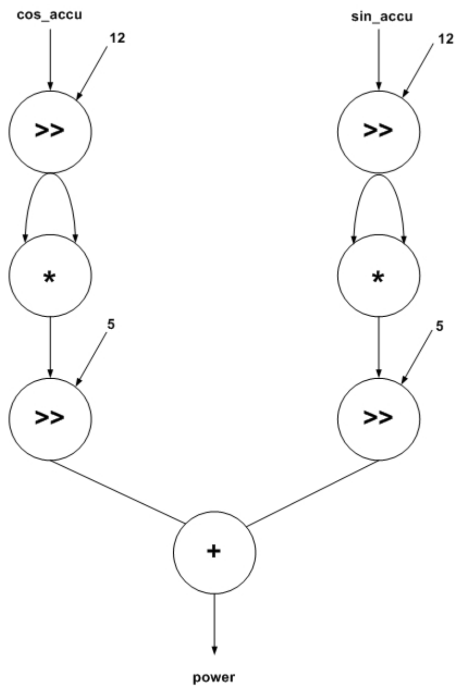


Figure 5.7: Square computations

DFT powers are computed. The done signal is asserted and it goes into the hold_val state, where it stays until another start signal is given. When the reset signal is asserted it always goes back to the idle state.

The last instruction is the `FSUBD` instruction. The control unit goes into the send state, where it sends the next two powers to the LEON in one cycle. This is done after the LEON has asserted the load signal. An internal counter keeps track of the powers to be send. After that it goes back into the idle state.

5.2.2 Binarization accelerator

As shown in Section 2.3.3 and Section 5.1.2 the binarization process uses a rotated grid, just like the DFT process. To enable the accelerator to compute all pixels in a block, not only the pixels of that block, but also the surrounding pixels of the rotated grid must be locally available. The worst-case scenario is depicted in Figure 5.8, the corner pixels in a block needs the most pixels outside the block.

From the figure it becomes clear that a 22x22 pixel block is needed in order to have all pixels locally available. It also needs a buffer (ROM) that contains the gridoffsets. So before the binarization accelerator can start its execution, it first needs 22x22 = 484 pixels to be send to it. This can be done with the `FADDD` instruction, which is shared with the DFT accelerator. The pixel buffer (RAM) is also shared, but now only 484 pixels are stored instead of 1156. As stated in Section 5.1.1, 128-bits can be send with

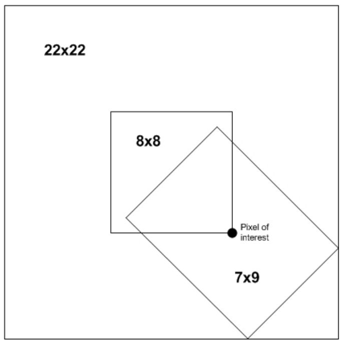


Figure 5.8: Size of the pixel block needed to perform the computations

a single `FADDD` instruction. To load all pixels in the binarization accelerator, (484 * 8) / 128 = 31 instruction calls are needed.

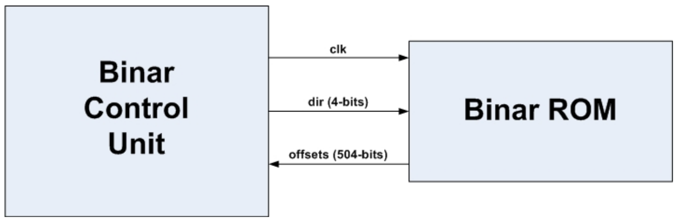


Figure 5.9: Block diagram of the binarization accelerator

The computations can be started with the `FDIVD` instruction. The first operand needs to contain the direction. The accelerator operates on one of the 16 directions, it is the responsibility of the software to send the correct direction. Normally the `FDIVD` instruction is used to divide two floating-point values with double precision. After computing the 64 binary pixel values, the first 8 values are returned by the `FDIVD` instruction. Normally a binary value is 1-bit and immediately all pixels could be returned with a 64-bit interface, but the software uses 0 and 255 as binary values. In order to save the compatibility and not having to include too much assembly code, it was decided to return the values as 8-bit values. This means 7 extra instruction calls are needed to get the other 56 values. It is a slightly slower solution, but results in a design that is much more clear. The `FSQRTD` instruction is used to send results back to the LEON. Normally this instruction is used to perform the square root on double precision floating-point values. In this case the `FSUBD` instruction could not be shared with the DFT accelerator, because the values are coming from a different buffer and other internal counters are used.

The binarization accelerator shares the control unit with the DFT accelerator. In Figure 5.1 this is depicted. The binarization accelerator is shown in this picture as the binarization execution unit, the DFT accelerator as the DFT execution unit. They also share the pixel block (RAM). Figure 5.2 shows the flow diagram of the control unit.

When there is a `FADDD` call, the control unit saves the pixels in the RAM. When there is a `FDIVD` call, the control unit asserts the start signal of the binarization execution unit. The control unit goes into the binar execute state and waits for the offset ready signal to be asserted by the execution unit. Offsets are needed to load the right pixels out of the RAM and into local buffers of the execution unit. It therefore goes into the binar load state, where it loads 63 pixels (rotation grid is 7 x 9 = 63) from RAM to buffers. As stated in Section 5.1.1 it is possible to load 4 pixels from the RAM in 1 cycle, so 16 cycles are needed to load the pixels. Then the control unit goes back into the binar execute state. It repeats the process until all 64 pixels are computed. When the `FSQRTD` call is received it goes into the binar send state, where it returns the next 8 pixels to the LEON.

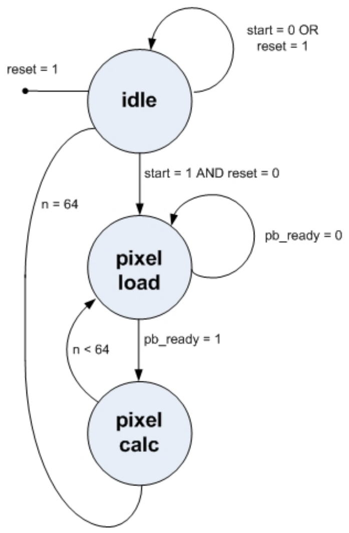


Figure 5.10: Flow diagram of the binarization control unit

Figure 5.9 shows the bock diagram of the execution unit. It consists of a control unit and a ROM that contains the rotation offsets for each of the 16 directions. Figure 5.10 shows the flow diagram of the control unit. There are only 3 states. After the start signal is given, a switch is made from idle to the pixel load state. Here it stays until all 63 pixels are loaded and the `pb_ready` signal is not asserted. After loading the pixels, the computations are performed in the pixel calc state. Because 64 pixels needs to be computed it goes back to the pixel load state and repeats the process 64 times. Afterwards it returns back to the idle state.

In the pixel load state the communication is handled between the accelerator and the control unit. Figure 5.11 shows the pixel calc state, where the computations are performed. In this state 72 byte additions are performed in only 1 cycle. Along with additions there are 72 counter increments, 10 comparisons and assignments. In total 64 pixels would take 64 cycles of computation time, but most of the time is spent on loading the pixels out of the RAM. In total 16 + 1 = 17 cycles are needed to compute

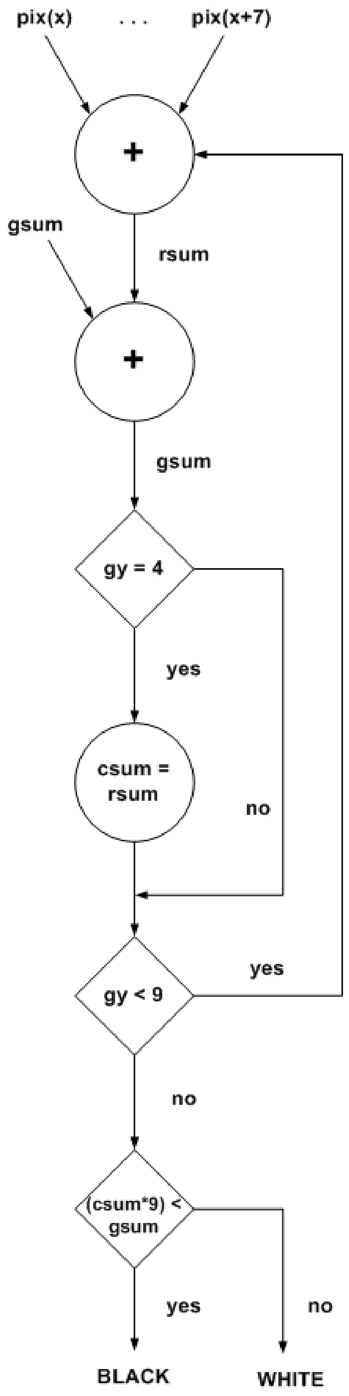


Figure 5.11: Computations performed in the pixel calc state

1 pixel. And 17 * 64 = 1088 cycles are needed to compute an 8x8 pixel block.

5.2.3 VHDL implementation

The complete co-processor consists of 7 VHDL source files. At the top-level there is the `co_proc.vhd` file. This contains the control unit and the two accelerator components, as shown in Figure 5.1. The control unit as shown in Figure 5.2 is implemented in this file. The third component that is port mapped in this file is the pixel block RAM. File `pbram.vhd` contains the implementation. The VHDL code is written in such a way that the synthesizer automatically infers and uses Block RAMs.

The DFT accelerator as shown in Figure 5.3 is implemented in `dft_co_proc.vhd` and contains a port map to the ROM modules. The sin and cos values are lookup-table values and stored in a ROM, implemented in `scrom.vhd`. The synthesizer can map ROMs onto Block RAMs for an efficient use of FPGA resources. The VHDL code is therefore written in such a way that this is recognized by the synthesizer. The other ROM, contains the gridoffsets and is defined in `gridrom.vhd`. This is the largest ROM of the co-processor and is therefore also mapped onto Block RAM.

The binarization accelerator shown in Figure 5.9 is implemented in `binar_co_proc.vhd`. It contains one port map to the ROM component that contains the offsets. It is defined in `binarrom.vhd`. Depending on how many Block RAMs are available on the FPGA, it can be decided to map this ROM (or any other ROM or RAM module) onto Block RAM, Select RAM or LUTs. This can be done by changing a flag, which is indicated in the source files.

5.3 Summary

In this chapter the hardware accelerators for speeding-up the MINDTCT system, are presented. A description for each accelerator is given along with some implementation details. The DFT power accelerator performs the most computation intensive part of the direction map generation process. It makes use of available block RAMs and MULs of the FPGA. The second accelerator presented is to speed-up the binarization process, where the image is converted from gray-scale to binary. Both accelerators are connected to the LEON by using the co-processor interface, this is the most efficient way for connecting them. Just like LEON the accelerators are developed completely in VHDL, are simulated to verify there correctness and are fully synthesizable. The next chapter shows the simulation and synthesis results.

Simulation and Synthesis 6

In this section the simulation and synthesis results will be presented. Simulation is performed to verify the output results of the accelerators and to determine there performance. Section 6.1 will present the performance comparison between the original system and the accelerated system. The LEON and the accelerators are synthesized for a Spartan3 FPGA, the results can be found in Section 6.2. In Section 6.3 a special application is presented that can communicate with the FPGA fingerprint system. It can be used to open any WSQ compressed fingerprint image, send it to the FPGA and display the computed minutiae. Finally in Section 6.4 some performance results are shown of the MINDTCT system executed on the FPGA.

6.1 Performance analysis

Simulation is not only performed to verify the output results, it can also be used for performance analysis. Simulating the system in `ModelSim` with the complete `MINDTCT` software is no option, a single fingerprint would take up to a week to simulate. The complete system is therefore simulated with `TSIM`, which reduces the simulation time to several minutes. The developers of the simulator state that it has an inaccuracy of about 10%, which is sufficient enough for the estimations. The accelerators could not be simulated in `TSIM`, because of the evaluation version limitations. They are simulated in `ModelSim`, which produces very accurate performance results. The simulation is performed for three iterations, which means computations are performed on three image blocks. This way the simulation time is not too long and a good estimate can be made.

Figure 6.1 shows the waveform for the DFT accelerator. Each iteration consist of a sending fase, an execution fase and a receive fase. In the first fase the pixels are loaded from memory into the accelerator. In the execution fase the accelerator is doing its computations. In the last fase the results are received by the LEON and stored back in memory. By performing 3 iterations a good estimate can be made, because all overhead between iterations is included. Figure 6.2 shows the same results for the binarization accelerator. From the figures it can be seen that three DFT iterations take 427.46 μs and one iteration takes 143 μs. Three binarization iterations take 113.24 μs and one iteration takes 38 μs.

To compare the performance with the original system, the amount of iterations needs to be determined. Table 6.1 shows the amount of iterations of `sum_rot_block_rows2()` and `dirbinarize_V2()`. It must also be determined how much time each iteration takes in the original system. These times can be determined by dividing the total time spend on each of the functions by the amount of function calls. The results can be found in Table 6.2. The data is collected for the first three images of the database and the images with the largest and smallest computation times.

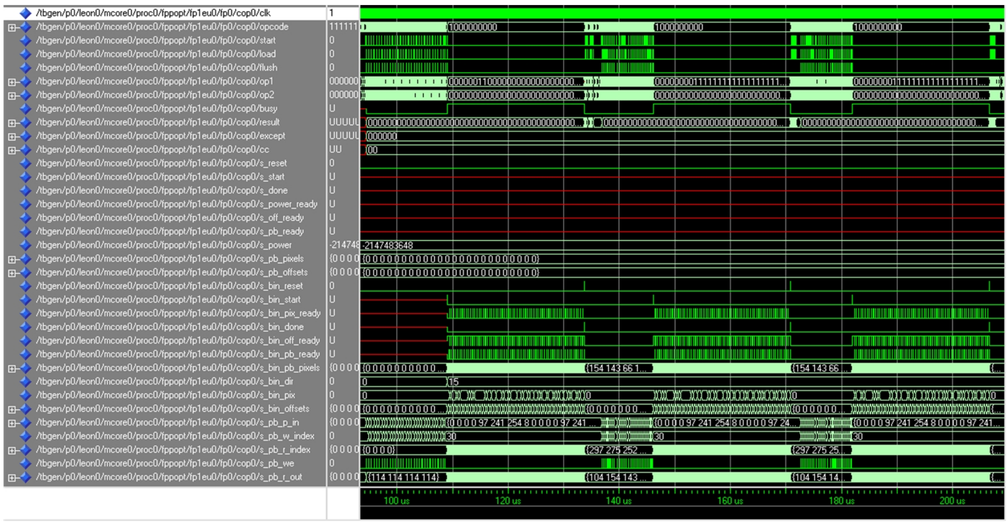


Figure 6.1: ModelSim waveform for 3 DFT iterations

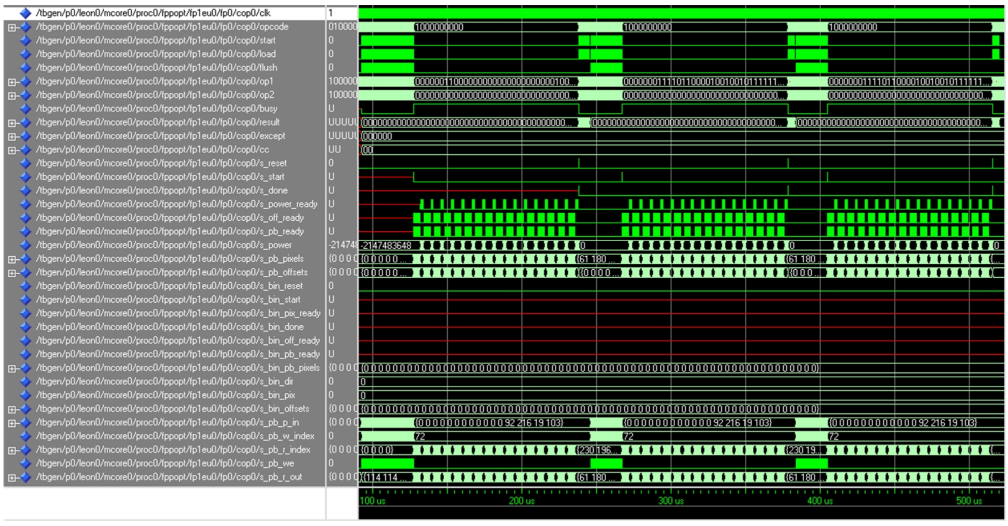


Figure 6.2: ModelSim waveform for 3 binarization iterations

From Table 6.2 it can be seen that the iteration times are almost the same for every fingerprint. This is because the computations are always the same and all block are of equal size. Only the amount of blocks (iterations) differ, as can be seen from Table 6.1. By multiplying the amount of iterations with the accelerator performance estimates for one iteration, the total amount of time can be derived. The total amount of time spend in the original system by the DFT and binarization steps where already derived and discussed in Section 3.5. Table 6.3 shows the comparison between the times spend by the software system and the accelerators.

	sum_rot_block_rows2()	**dirbinarize_V2()**
Fingerprint1	2196	1981
Fingerprint2	2479	2262
Fingerprint3	2119	1734
Fingerprint6	2497	2200
Fingerprint9	1607	1188

Table 6.1: DFT and binarization iterations

	sum_rot_block_rows2()	**dirbinarize_V2()**
Fingerprint1	$1.51\text{x}10^{-3}$	$9.04\text{x}10^{-4}$
Fingerprint2	$1.51\text{x}10^{-3}$	$9.07\text{x}10^{-4}$
Fingerprint3	$1.51\text{x}10^{-3}$	$9.05\text{x}10^{-4}$
Fingerprint6	$1.51\text{x}10^{-3}$	$9.05\text{x}10^{-4}$
Fingerprint9	$1.51\text{x}10^{-3}$	$9.01\text{x}10^{-4}$

Table 6.2: Execution times (seconds) for 1 iteration

From Table 6.3 it can be seen that the accelerators speed-up the process by a significant amount. For a better performance overview the execution times and the effect of the accelerators on the complete process must be determined. Table 6.4 shows the execution times of the complete system when using only the DFT accelerator, only the binarization accelerator or using both accelerators. It can be seen that the DFT accelerator gives more speed-up than its binarization counterpart. Combined they will speed-up the system by a large factor.

Table 6.5 shows the improvements in seconds and percentages. The average execution time of the system without accelerators is 16.63 s. When using the accelerators it is reduced to an average of 12.04 s. The DFT accelerator shows an average improvement of 17.77% and the binarization accelerator an average of 9.63%. This means the complete system is improved by an average of 27.4%. The efficiency of the accelerators is very high, looking at the theoretical maximum improvement of 29.68% explained in Section 3.5. It should also be noted that the total amount of speed-up is much higher, because the times from the tables above include the software improvements. Table 6.6 shows the execution times of the original system without the DFT and binarization software modifications and the accelerated system. It can be seen that both the software modifications and the accelerators are speeding up the system by an average of 7.42 s or 37.91% and a maximum of 8.57 or 41.06%.

6.2 Synthesis results

The `LEON` micro-controller is highly configurable, and therefore the best configuration needs to be found. This effects the size of the FPGA implementation. The `LEON` is kept as small as possible, to limit the resource usage. The `leon2-1.0.32-xst` version is

	DFT soft	**binar soft**	**DFT hard**	**binar hard**
Fingerprint1	3.31	1.79	0.31	0.08
Fingerprint2	3.73	2.05	0.35	0.09
Fingerprint3	3.19	1.57	0.30	0.07
Fingerprint6	3.76	1.99	0.36	0.08
Fingerprint9	2.42	1.07	0.23	0.05

Table 6.3: Comparison between software and hardware times (all in seconds)

	DFT acc	**binar acc**	**both acc**
Fingerprint1	13.96	15.25	12.25
Fingerprint2	14.26	15.68	12.30
Fingerprint3	13.43	14.82	11.93
Fingerprint6	14.62	16.11	12.71
Fingerprint9	12.01	13.18	10.99

Table 6.4: Execution times (seconds) for complete system when using the accelerators

used for the synthesis. It comes with a complete FPU interface and is one of the latest version of the `LEON2` model. The `LEON3` version is providing better performance, but lacks an usable FPU interface as described in Section 4. The target FPGA board is the Xilinx Spartan3 XCS2000, which is the board that is available on the CAS laboratory of the TU Delft.

With the `xconfig` call an graphical interface can be started to configure the `LEON`. In case of this project many options were disabled. First the Virtex2 libraries are used to do the synthesis, these are compatible with the Spartan3. Then a macro is enabled to re-synchronize the clock and improve the clock-to-output delays. Also it divides the clock by two. On the FPGA three oscillators are available, a 100 MHz, 66 MHz and a 40 MHz. The `LEON` without the accelerators can run on a 33 MHz clock, that can be created by using a Digital Clock Manager (DCM) that divides the 66 MHz clock by two. An even higher clock rate is possible as can be seen from Table 6.7. When the accelerators are synthesized the clock needs be lowered to 25 MHz. This is due to a strange behavior of the LEON architecture, even when a very simple and fast co-processor is connected the critical path is increased and the clock rate must be lowered. The macro also re-synchronizes the SDRAM clock. When the macro is not used the inverted clock signal option needs to be enabled to get a synchronizes SDRAM clock.

The divide and multiply units needs to be enabled along with there instructions. Also the software needs to be compiled with the `-mv8` flag to enable the `div` and `mul` units. The multiplier is set to a 5 cycle latency, which gives the best area/timing/performance compromise. A 16x16 multiplier with a pipeline registers is then inferred by the synthesizer. For the standard configuration the co-processor and FPU needs to be

	org (s)	DFT impr (s)(%)	binar impr (s)(%)	total (s)(%)
Fingerprint1	16.96	3 (17.69)	1.71 (10.08)	4.71 (27.77)
Fingerprint2	17.64	3.38 (19.16)	1.96 (11.11)	5.34 (30.27)
Fingerprint3	16.32	2.89 (17.71)	1.50 (9.19)	4.39 (26.90)
Fingerprint6	18.02	3.40 (18.87)	1.91 (10.60)	5.31 (29.47)
Fingerprint9	14.20	2.19 (15.42)	1.02 (7.18)	3.21 (22.60)

Table 6.5: Improvements shown in seconds and percentages

	original (s)	mod+acc (s)	impr (s)	impr (%)
Fingerprint1	19.55	12.25	7.30	37.34
Fingerprint2	20.87	12.30	8.57	41.06
Fingerprint3	19.18	11.93	7.25	37.80
Fingerprint6	21.27	12.71	8.56	40.24
Fingerprint9	16.43	10.99	5.44	33.11

Table 6.6: Comparison between original and modified and accelerated system

disabled. The software needs to be compiled with the `msoft-float` flag, so no FPU instruction are present in the assembly code. In fact when the accelerators are used, the software still needs to be compiled this way to avoid incorrect usage of the accelerators.

In order to reduce power-consumption and gate-count, a direct-mapped cache is used. The associativity is therefore set to 1 set with a size of 8 KB and a line size of 32 bytes. The data cache is kept the same as the instruction cache. Different cache sizes and implementations effects the performance of the system, but a study on this is out of the scope of this project.

Large units that occupy many resources and are not needed, are all disabled. Examples are: the Memory Management Unit (MMU), Ethernet interface, PCI controller, trace buffer, etc. The Debug Support Unit (DSU) is enabled to allow non-intrusive debugging. The software can then be loaded into the implementation on the FPGA by using the DSU UART interface and connect to it by using the `Dsumon` program. Also the SDRAM controller is included to be able to use SDRAM memory. This memory is needed, because of the size of the `MINDTCT` software system. It does not fit into internal memory or the SRAM memory provided by the FPGA.

The `LEON` is synthesized for the Xilinx Spartan3 XCS2000 with FG676 package and a speed-gate of -5. The synthesizer that is used is Synplicity Synplify Premier 8.6.2 with the FSM compiler, Resource Sharing, Pipelining and Re-timing options turned on. The FPGA performance and resource usage results for the `LEON2` without accelerators can be found in Table 6.7. When the accelerators are also synthesized the FPU unit must be enabled via the graphical interface. It must be set to the LTH interface and in `device.vhd` some modification are needed. In this file the FPU interface is always set to serial, it must be changed to parallel in order to get much better performance. In this case the critical path extension is limited and accelerator instructions operate in parallel with the Integer unit. The performance and resource usage results can be found in Table 6.7. It can be seen that the accelerators take up as much Block RAMs as possible, in order to keep the LUT usage as low as possible and make use of the resources available on the FPGA. Only 4 extra Block MULs are needed to perform the multiplications in the DFT accelerator. The accelerators take up 7506 LUTs, which is very much equal to the size of the `LEON`. The internal clock speed is limited to 27.2 MHz, due to the behavior of the `LEON` architecture as explained before. The system created in this project forms the basis of a low-power fingerprint chip that is the target of future projects. Such a chip requires an clock speed that is much lower than 27.2 MHz. Therefore it is no problem that the clock speeds are lower, the system with accelerators will run about 40% faster than the stand-alone `LEON` running on the same clock speed.

	LEON2	**LEON2 with accelerators**
clk	79.1 MHz	54.3 MHz
clk/2	39.6 MHz	27.2 MHz
Register bits	2257 (5%)	7222 (18%)
Block RAMs	12/40 (30%)	39/40 (97%)
Block MULs	1/40 (2%)	5/40 (12%)
Clock Buffs	2/8 (25%)	2/8 (25%)
Total LUTs	7052 (17%)	14558 (35%)

Table 6.7: Synthesis results for LEON2 with and without the accelerators

6.3 Fingerprint application

In order to test the system in terms of minutiae extraction results instead of performance, an applications is created to show the results. It can open any WSQ compressed fingerprint image and show it on the screen. It contains the same WSQ decoder as used in the `MINDTCT` software. When the print is shown on the screen the user has the ability to zoom-in to have a more clear view on the bifurcations and terminations of the ridge segments in the fingerprint.

The applications can communicate with the `MINDTCT` hardware system by using the UART interface. The `MINDTCT` system can run in the `TSIM` simulator and the application can make contact with it by using a virtual COM port simulator. This way the system could be tested in the simulation fase. With `TSIM` it is not possible to set the UART parameters such as: baudrate, data bits, etc. It detects and uses the latest settings of the UART. The applications needs to be started before the simulator is started, it will then initialize the UART with the right values. The application uses COM3 by default and TSIM needs to be send with the `-uart1 //./COM2` option in order to use COM2. Also the COM2 and COM3 connection needs to be emulated.

The WSQ fingerprint data is send to the simulator. The application and the simulator perform some handshaking in order to startup the sending and receiving. When all the data is received by the `MINDTCT` system, it starts its execution. The applications waits and polls the UART for a finish signal. When this is received it starts collecting the minutiae list, it displays a dialog box that shows the amount of minutiae computed. Then it marks all computed points in the fingerprint by showing them as red dots. Figure 6.3 shows a screen capture of the application with the fingerprint image and Figure 6.4 shows the same fingerprint with the computed minutiae by the `MINDTCT` system.

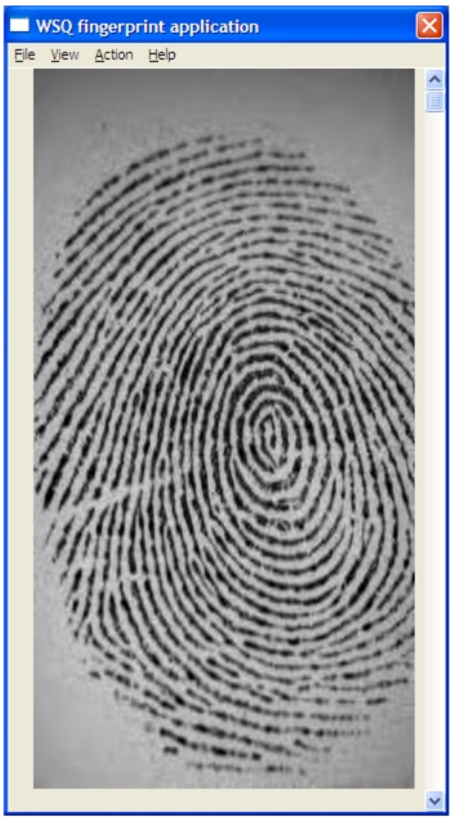


Figure 6.3: Application showing a fingerprint image

6.4 FPGA results

After the synthesis with Synplicity, a UCF file is created. It specifies the constrains and pin assignments. Basically it connects the top-level VHDL entity signals with the right pins on the FPGA. In order to get the SDRAM module to work several things are needed. First the `LEON` needs to include the SDRAM controller, with the

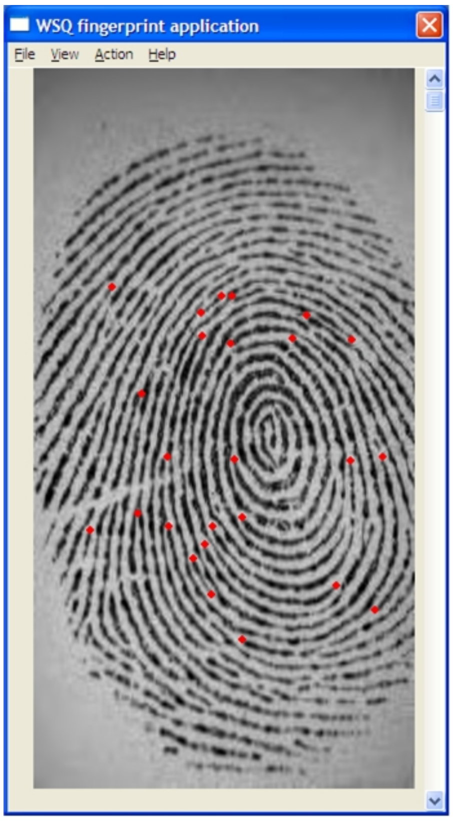


Figure 6.4: Application showing a fingerprint image with minutiae computed by the MINDTCT system

separate address and data bus option and inverted clock turned on. Then all signals that has to do with the SDRAM unit needs to be port-mapped in the top level entity in `leon_spartan3.vhd`. In the UCF file these signals needs to be mapped to the right pins on the FPGA, corresponding to the manual of the SDRAM module. The module needs to be inserted on the P3 port of the Spartan3 FPGA board.

Two switches are needed in order to be able to put the `LEON` into debug mode. These switches needs to be mapped to the DSU enable and break signals of the top-level entity. When the `LEON` is startup without an application, the fastest way to load an application into it, is to use `Dsumon`. This applications can connect to the FPGA via the UART interface and detect the configuration. Then with the `lo` command an application compiled with the SPARC compiler, can be loaded into the FPGA. Execution of the software can be started with the `run` command.

Table 6.8 shows the results when the `MINDTCT` system is executed on the FPGA. It can be seen that the performance is not as high as in the simulator. There are several reason for this. The first one is the slower clock rate of 33 MHz. The simulator runs on

50 MHz, which means the execution time is already 1.5 times higher. But there still is a large difference remaining, even if the clock rates where equal. This is mainly due to the limited emulation options of the evaluation version of `TSIM`. The simulator can only be configured with an unrealistic and much faster memory interface. The SDRAM option could not turned on in the simulator, as with other advanced settings for memory and caches. Also the simulator internally simulates the `LEON3` model instead of the `LEON2`. In order to simulate the `LEON2` an more older version of `TSIM` had to be used, which is no longer supported and available. Also the simulator has an inaccuracy of 10%, on top of the `LEON3` simulation this means the inaccuracy is increased to more than 30%. Also the execution is controlled by the `Dsumon` program, and this means that communication overhead is included in the execution times. Simulating in `ModelSim` will provide a far better estimation of the real FPGA execution time, but has the drawback of being too slow, when simulating programs with the size of the `MINDTCT` system. The accelerators by them self are simulated using `ModelSim` so these estimations should be close to the real values.

	LEON2	**LEON2 with accelerators**
Fingerprint1	97.58 s	61.14 s
Fingerprint2	105.23 s	62.02 s
Fingerprint3	93.29 s	58.03 s
Fingerprint6	105.88 s	63.27 s
Fingerprint9	83.29 s	55.71 s

Table 6.8: Performance results of the MINDTCT system running on the FPGA

Although the accelerators run perfectly fine in the simulator and can be synthesized, there are some problems with them. It seems that on the FPGA they run as expected, but only zeros are written back to memory. Probably there are some timing problems. Due to limited time, it was not possible to fix this problem and only the simulations estimations are available. On the right side of Table 6.8 these estimations are shown.

6.5 Summary

In this chapter the simulation and synthesis results are presented. Simulation shows that the accelerators and algorithm modifications will speed-up the system by almost 40%. Synthesis shows that the accelerators use little resources, but effect the clock-rate. Due to a strange behavior of the `LEON` architecture, the critical-path is increased when a co-processor is connected to it. Even if the accelerator is very simple and very fast. Also a short description of an application is given that can be used to show the output results of the MINDTCT system. After synthesis the system is implemented on a FPGA. Execution times are shown to be much higher on the FPGA than on the simulator. This is due to simulator inaccuracy and limitations. Despite the performance issues, the system runs perfectly fine. The next chapter presents the conclusion and future work for this project.

7 Conclusion and Future work

In this last chapter, the conclusion is presented in Section 7.1 and future work in Section 7.2.

7.1 Conclusion

In this thesis a fingerprint minutiae extraction system is created that runs on a FPGA. It forms the basis for a fingerprint identification or verification chip. The system is based on the System-on-Chip (SoC) principle. It is a combination of hardware and software. The hardware part consist of an open-source microcontroller completely available in VHDL, named `LEON2`. The software is a modified version of the `MINDCT` system, originally created for the FBI by NIST.

The first step was to find the best suitable software system for this project. A minutiae extraction system consists of many algorithms. The target was to find a collection of good performing algorithms, that forms a complete system. Because of the nature of the algorithms, many where only available in Matlab code. Experiments showed that converting Matlab code to C-code is resulting in very poor performance and terrible looking code. The target therefore was reset to find a complete minutiae extraction system completely available in C-code. At the time there was only one useable system, called `MINDTCT`. This system however is developed to be executed on a PC, and needed to be modified so it could run on a microcontroller. It had to be stripped down, which means all non essential code had to be removed. Also to increase performance and save resources, all floating-point instructions needed to be converted to fixed-point counterparts.

The second step was to find a good performing microcontroller, that can run the `MINDTCT` program. Although many controllers are available, research showed that the `LEON` microcontroller was favorite. With the right controller found, the software system could further be modified so it can run on this controller.

With the right hardware and software found, the first part was to build a minutiae extractor that can run on a FPGA. The second part was to speed-up the execution of this system. The `LEON` not only has good performance, it is also possible to connect co-processors to it. In order to build efficient co-processors the software was profiled and the most time consuming parts or functions where identified. This resulted in two candidates that could be mapped onto hardware. The accelerators where connected to the `LEON` by the fastest interface that was available, called the CPI interface.

The system is simulated in a special program called TSIM, which is an instruction-set simulator. Simulating the system in ModelSim would take up to one week for a single fingerprint. With TSIM this is reduced to minutes. The system was tested on its output and performance analysis was performed. In order to get a more clear view of the minutiae output results of the system, a special applications was created that can show the computed minutiae in a fingerprint. Minutiae are displayed by red dots in this application. The modified system running on the FPGA produces almost the same minutiae as the original system. Only a few with low quality factors are missed or introduced, but these will be ignored in the matching step.

The system is implemented on the Xilinx Spartan3 XCS2000 FPGA. Performance analysis on the FPGA shows that the results of the TSIM simulator are unrealistic and too optimistic. This was due to limited emulation option of memory and caches. In order to get more realistic results an license needs to be purchased. Furthermore the simulator simulates the newer LEON3 system, which results in an inaccuracy of more than 30%. Despite the lower performance the system runs perfectly fine on the FPGA.

The accelerators run fine in the simulator and can be synthesized, but when executed on the FPGA there are some problems. It seems that they run as expected, but only zeros are written back to memory. Probably there are some timing problems. Due to time problems, this could not be fixed on time. However the simulation is done in ModelSim, which should give much better estimates than the TSIM simulator. The system with accelerators and software modifications speed-up the system by an average of 37.91% and a maximum of 41.06% for the 30 fingerprint from the test database. The system with accelerators can run on a speed of up to 27.2 MHz. The LEON without accelerators can be clocked on a slightly higher clock speed. This lower clock speed comes from behavioral aspects of the LEON architecture. When a co-processor is connected the critical path is increased, no matter how fast the co-processor can operate.

This project shows it is possible to run a fingerprint minutiae extractor on a FPGA, while using limited resources. Because the software is very computationally intensive, it takes an average of 60 seconds to complete one fingerprint. The accelerators will speed-up the system by almost 40%. The solution is based on the System-on-Chip (SoC) principle and therefore provides a basis for a fingerprint chip.

7.2 Future work

The accelerators only work in the simulator. Debugging and testing is needed, to identify the problem and making them to work on the FPGA. Then performance analysis, on the FPGA needs to be performed. To get a full identification or verification system, also the matching part is needed. Besides the MINDTCT system, also a matching system is developed by NIST. This system needs the same modifications such as: stripping and fixed-point conversion. Profiling should be performed, to identify if accelerators can be used to speed-up the execution.

Because this project was created to provide a basis for an identification chip, there is a lot of room for improvements. There are two parts where improvements can be made. The first is the software system. Profiling shows that some algorithms are responsible for a large part of the execution time. In this project it was shown that modifications can speed-up the system. Further improvements must be made on the algorithms, especially the DFT and binarization parts. One should try to lower the amount of iterations, by using more clever algorithms. This will increase performance by a large amount. The `MINDTCT` system contains many different algorithms, which could be replaced by better ones. Different algorithms can provide better extraction results, better performance or both. The `MINDTCT` system should also be re-written for execution on a micro-controller, this should results in better memory utilization, less wasting of resources and faster computations.

The second part where improvements can be made are on the hardware system. At the time `LEON2` was the best option available, but this could be different in the future. The `LEON3` is faster and more efficient, but lacks the CPI interface. It is needed to connect accelerators in a fast way. It is possible to extend the `LEON3` with such a interface, but this requires knowledge of the implementation of the microcontroller especially the Integer pipeline. Doing this will result in a system that has an performance increase of about 30%. Also higher clock rates are possible. This project shows that execution times are high on a FPGA, so the system must be accelerated more than 30%. A very good option for such a performance increase, would be to search for a very simple and basic microcontroller and adapt it to run the software. The most important thing would be the memory architecture. If both the microcontroller and the accelerators can access the fingerprint image data in memory, the communication overhead will be reduced. This creates far more opportunities for doing computations in hardware. For example, the whole DFT and binarization processes can be done in hardware. Depending on the resources even the whole minutiae extraction can be performed with accelerators, which on the `LEON` takes more than 70% of the time. In short, to get a significant performance increase the algorithms should be modified, the software adapted for execution on a micro-controller and a better memory architecture needs to be found, in order to be able to shift more computations to hardware.

Bibliography

[1] Govindaraju V. Chikkerur S. and Cartwright A.N., *Fingerprint image enhancement using stft analysis*, Pattern Recognition (2007).

[2] Mattsson D. and Christensson M., Evaluation of synthesizable CPU cores.

[3] F. Galton, *Finger prints*, McMillan, 1892.

[4] Wan Y. Hong L. and Jain A.K., *Fingerprint image enhancement: Algorithms and performance evalution*, IEEE Transactions on Pattern Analysis and Machine Intelligence (1998).

[5] SPARC International Inc, The SPARC Architecture Manual.

[6] Davide Maltoni and Dario Maio, *Handbook of fingerprint recognition*, Springer-Verlag, 2003.

[7] Studies of Fingerprint Matching Using the NIST Verification Test Bed (VTB), Test results for analysis on the VTB system using the NIST MINDTCT extractor.

[8] National Institute of Standards and Technology (NIST), User's Guide to NIST Fingerprint Image Software (NFIS).

[9] Gaisler Research, GRLIB IP Library User's Manual.

[10] Monro D.M. Sherlock B.G. and Millard K., *Fingerprint enhancement by directional fourier filtering*, Visual Image Signal Processing (1994).

[11] J.H. Wegstein, *An automated fingerprint indentification system*, U.S. Government Publication, 1982.