A Fast and Low Cost SIMD Architecture for Fingerprint Image Enhancement

MSc THESIS

Mian Qin

Advisor: Dr. Rene van Leuken

October 2005

Microelectronics

Department of Electrical Engineering

Faculty of Electrical Engineering, Mathematics and Computer Science

Master thesis supervisor/advisor:

Dr. René van Leuken

Exam committee:

Prof. Dr. ir. Patrick Dewilde

Dr. ir. René van Leuken

Dr. ir. Nick van der Meijs

Dr. ir. Arjan van Genderen

Special thanks:

Dr. René van Leuken

Abstract

As fingerprint verification system (FVS) grows more and more mature, it becomes the most popular and reliable biometric identification technique. Fingerprint image enhancement is a quite important step in FVS. The reliability and accuracy of the minutiae extraction mainly depend on a strong enhancement algorithm. Also it dominates the computational complexity of the whole system since it involves a huge number of convolutions. Earlier, Hong and Jain proposed a fingerprint image enhancement algorithm based on the Gabor filter, which has been proved to be quite an efficient and reliable method to improve the quality of fingerprint images. There are a lot of studies on applying this algorithm on a general purpose processor. However, because it introduces too many multiplications, the processor on a mobile embedded device may seem to be unsatisfying to achieve the goal of real time. Otherwise, there is a pain in losing the accuracy when lowering the computational complexity.

This thesis is focusing on developing an efficient hardware architecture which greatly speeds up the enhancement processing for mobile embedded devices. Based on the characteristics of the algorithm, a parallel architecture is introduced, which use a SIMD (single instruction streams and multiple data streams) structured processor array. On theory, the speed will be boosted by n times, where n is the number of processors. But due to the limitation in data transmission speed and overloads in parallel structure, the speed gain is less than n. And also a technique called data pre-fetching is used to release the bottleneck of huge data transmission between external memory and processors. In the best case, this technique is able to twice the processing speed with an acceptable hardware cost. With the help of fast hardware multiplier a multiplication instruction can be executed within one clock cycle. The specialized processor architecture and instruction set make it possible to incorporate large accumulation operations into a single instruction and gains in speed.

Because there are no hardware floating-point units (FPU) in this architecture, fix-point representation is used. This saves large hardware areas as well as time cost on floating point operations with a negligible accuracy losing. Moreover, this architecture can be also efficient for many other algorithms in image processing. And due to the flexibility in architecture, it can be adjusted to meet different hardware constrains. Experiment results show that this architecture greatly improves the speed with acceptable hardware area.

CONTENTS

List of Figures

List of Tables

Acknowledgements

First, I would like to thank my advisor Dr. Rene van Leuken, for his great help and advice in my thesis. The most important thing I've learned from him is the way to look into a problem and the attitude in research which also helps in everyday life. My thanks also go to the colleague students in the Lab for all the discussions.

And great thanks for my parents!

Chapter 1 Introduction

Fingerprints have been used in biometric identification verification since ancient times. This is because fingerprints have 5 advantages overcoming other biometrics techniques: a. UNIVERSAL, b. UNIQUENESS, c. PERMANENCE, d. COLLECTABILITY, e. ACCEPTABILITY. With these characteristics, fingerprints are widely used for identity verifying and security purpose.

But it's only in recent years that highly reliable and real time Fingerprint Verification System (FVS) becomes possible. With the help of fast computers and large storage devices, a huge database of fingerprints from different persons can be established and reliable algorithms can be used for matching and indexing in the database.

While the algorithms grow more and more mature, the FVS becomes more and more widespread in applications. However, it's not satisfying to both meet the high reliability and real-time requirements when applying FVS on mobile embedded devices. This thesis focuses on analyzing and giving a solution to this problem.

This chapter is organized as follows. Section 1.1 introduces the background of FVS. Section 1.2 gives the motivation on this thesis. Section 1.3 generally describes the purpose we expect to achieve. Section 1.4 lists the framework of this thesis.

1.1 Background of FVS

A fingerprint verification system can give a complete solution on fingerprint collecting, analyzing and matching, widely applied for criminal investigation, authentication and security purposes. Typically, a FVS consist of four parts: fingerprint input, enrollment, authentication and database, which are shown in Figure 1.1.

The input fingerprint image can be obtained by different kinds of scanners, such as optical scanner as well as capacitance scanners. In Enrollment module, the features of input fingerprints, called minutiae, are extracted and stored in the database while the Authentication module tries to find a matching one in the database for the input fingerprint. As a matter of fact, the Authentication module first does the same job as Enrollment module to extract the minutiae in the input image and then matches with every record in the database to find out the matched one.

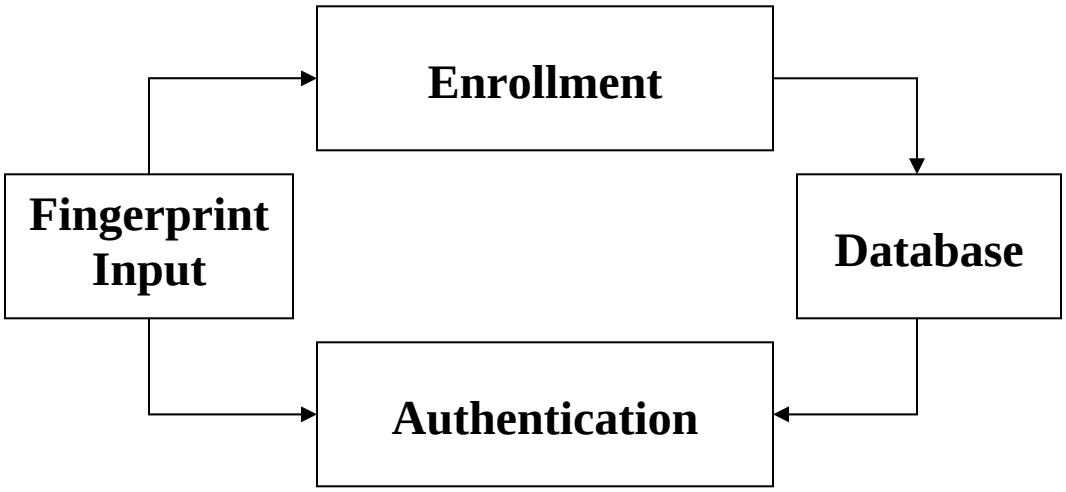


Figure 1.1 Architecture of FVS

Now it's clear that there are mainly two major tasks: minutiae extracting and matching. There are many studies on these two subjects, proposing many different algorithms and many efficient and reliable algorithms become available. Despite what methods they use, they may face the same problem that in practice the quality of input image may be quite poor due to predictable factors such as thin ridges, scars as well as unpredictable factors such as dry/wet fingers, moving fingers and as a result to produce many incorrect minutiae. These incorrect minutiae may greatly degrade the accuracy and reliability of the FVS. In order to overcome this drawback, it's necessary to introduce image enhancement before minutiae extracting.

Among the all, the algorithm based on the Gabor filter proposed by Hong and Jain [1] is proved to be quite efficient in improving reliability of FVS. This approach involves huge numbers of convolutions, which can be computationally expensive.

1.2 Motivation

In our daily lives, it's more and more desired to incorporate FVS into mobile embedded devices. But because the following two reasons, there is always a limitation or even a skip in the important step of fingerprint image enhancement: a) huge computational expense, b) lack of floating-point processing units (FPU).

By adjusting the convolution size and algorithm parameters, the computational expense can be reduced to meet the requirement of real-time. But it also degrades the improvement of image quality. It's always a dilemma when applying enhancement algorithms on weaker general purpose processor (GPU). Another embarrassment is that most embedded devices are equipped with RISC processors such as the ARM and StrongARM processors without hardware floating-point processing units (FPU) while fingerprint image enhancement procedures usually require substantial quantities of floating-point operations. Such omissions can cause downgrades of the performances of most fingerprint verification systems when handling low quality fingerprint images.

Thus, it's really a challenge for us to apply the FVS under extreme hardware constrains, while still keeping the accuracy and high speed.

1.3 Thesis goals

As stated before, designing an efficient hardware implementation for fingerprint enhancement becomes desired and urgent. Three goals of this thesis are listed in the following.

- **This hardware implementation can be applied on smart card system with low power consumption.**
- **This implementation should give a real-time solution for fingerprint enhancement.**

1.4 Conclusion

In this thesis, a SIMD structure is given. This implementation requires acceptable hardware area with a scalable architecture. And due to the lack of FPU, the power consumption is also reduced. Thus the first goal is achieved.

Further, this architecture is implemented in VHDL and the simulation results prove that it has a reliable performance in real-time application.

The optimization in architecture and synthesis of the VHDL code will be the future work to apply this architecture on Xilinx Spartan 3 XC3S2000 chip.

1.5 Thesis overview

The rest of this thesis is organized as follow:

Chapter 2 discusses the existed algorithms in fingerprint enhancement. After analyzing and choosing the best algorithms, some modifications are also made to improve the enhancement.

Chapter 3 gives the software implementation on these algorithms in order to validate the feasibility and improvement of the work in chapter 2.

Chapter 4 focuses on architecture study base on the fingerprint enhancement algorithms and proposes a SIMD architecture for real-time application on smart card system.

Chapter 5 describes the VHDL implementation of the SIMD architecture proposed in chapter 4. More details in this architecture will be explained and implemented in this chapter.

Chapter 6 is about the simulation phase, with result analysis.

Chapter 7 summarizes the work draws a conclusion. A future research direction is proposed.

.

Chapter 2 Related work

The fingerprint enhancement algorithm based on Gabor filer was first proposed by Hong, and later on many improvements and optimizations were made by Hong and also others [1], [2], [3], [4]. As a result, the enhancement is emphasized by elaborately selecting filter parameters as well as the computational complexity is released without losing notable accuracy. Base on his algorithms, I introduced two new steps, Gaussian filtering and local normalization to cope with degradation due to the hollows and discontinuities presented in the fingerprint ridges.

This chapter is organized as follows: section 2.1 presents backgrounds in fingerprint verifications technique. Section 2.2 proposes an efficient fingerprint enhancement algorithm base on Hong's work, and also gives the algorithms analysis in details. Section 2.3 summarizes this chapter.

2.1 Background

Since a person was born the fingerprints will keep unchanged over a lifetime. Fingerprints are distinguishing, which means every person has differences in their fingerprints and the differences can be technically distinguished and used as a personal identification.

Fingerprints are graphical flow-like ridges present on human fingers. By some acquisition devices such as optical scanners or capacitance sensors, we can capture a fingerprint in a gray scale image. Thus a fingerprint can be represented by the entire gray scale profile, which is called ridge based representation. However such applications will be limited by

some factors such as brightness, variations, image quality variations, scars, and large global distortions present in fingerprint image. The landmark based representations [3] described below can overtake such drawbacks because it is easier to verify in principle. Another reason for landmark representation is for security because it is impossible to reconstruct the entire fingerprint image only from the landmark information.

The common hypothesis underlying such representations is the belief that the individuality of fingerprints is captured by the local ridge structures (minutiae details) and their spatial distributions. Although there are more than one hundred types of local ridge structures, we mainly rely on two most prominent structures: ridge endings and ridge bifurcations.

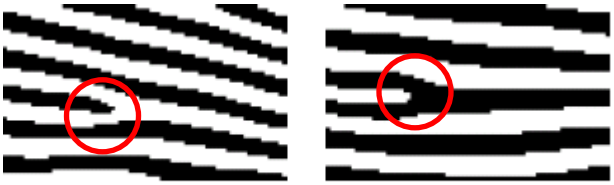


Figure 2.1 Example of ridge endings and ridge bifurcations

Thus, extracting minutiae from a fingerprint image plays an essential role in the FVS.

2.2 Fingerprint image enhancement

2.2.1 Minutiae extraction and image enhancement

Correctly extracting the minutiae information from the fingerprint image guarantees the reliability of FVS. However, if the minutiae information on the image is spurious somehow or even missing, how can the correct minutiae be extracted? In practice, this is possible due to unpredictable reasons such as physical damage on fingerprint, sweat, or moving fingers, etc.

In figure 2.2, in the red blocks, new spurious minutiae may be detected or real minutiae may be missing by the extracting algorithm. While in figure 2.3, it's clear and easy to

find each minutia just as in these blue blocks. That's to say, the minutiae information are well preserved in the image.

For those images like in figure 2.2, it's necessary to do image enhancement before minutiae extraction. This may greatly restore the missing information and remove spurious minutiae. In figure 2.4, the image in the left is obtain by a capacity scanner, while in the right side, the enhanced image shows a good improvement in the quality and is then ready for the further extraction.

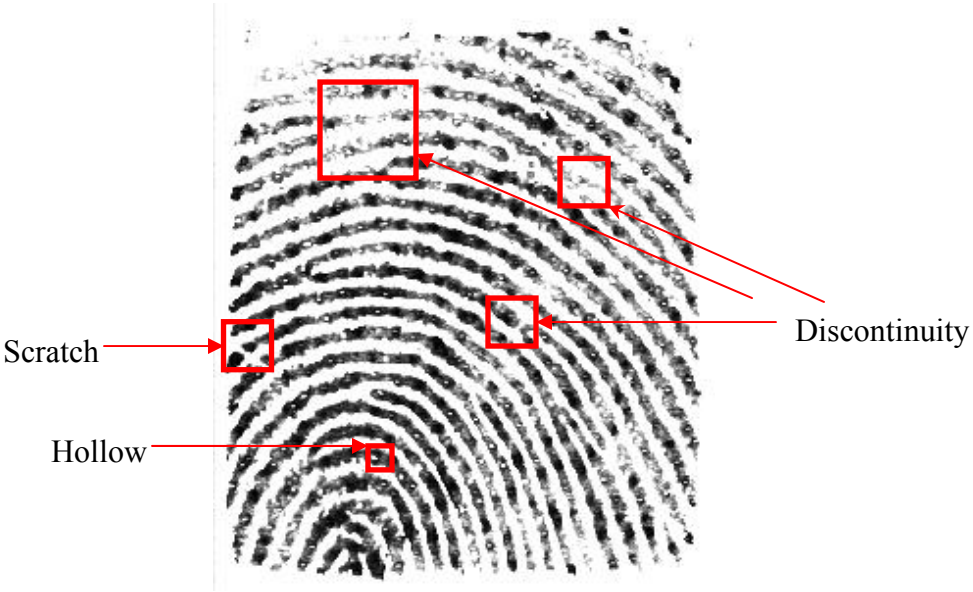


Figure 2.2 Fingerprint image in poor quality

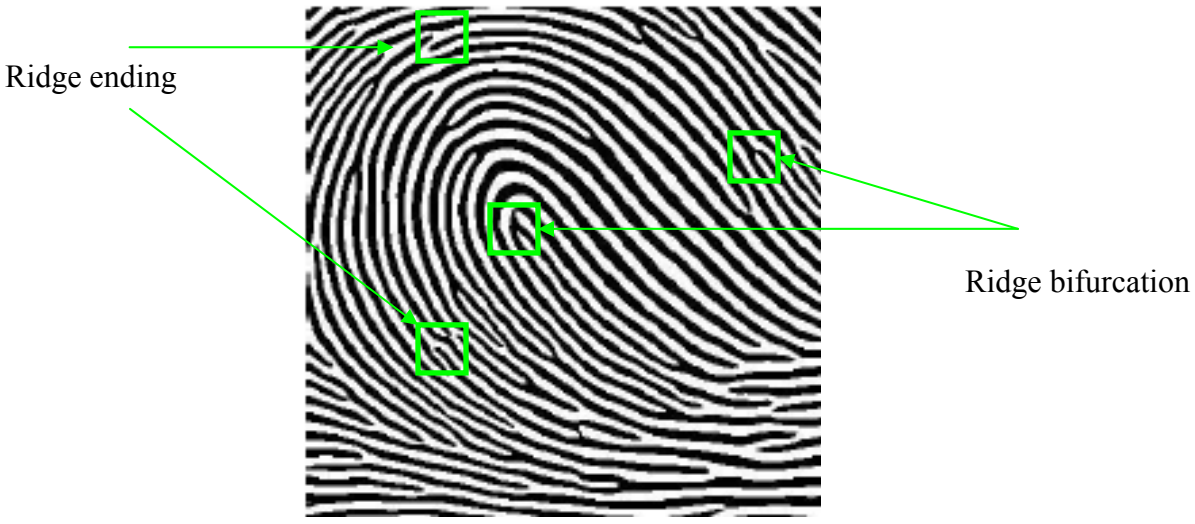
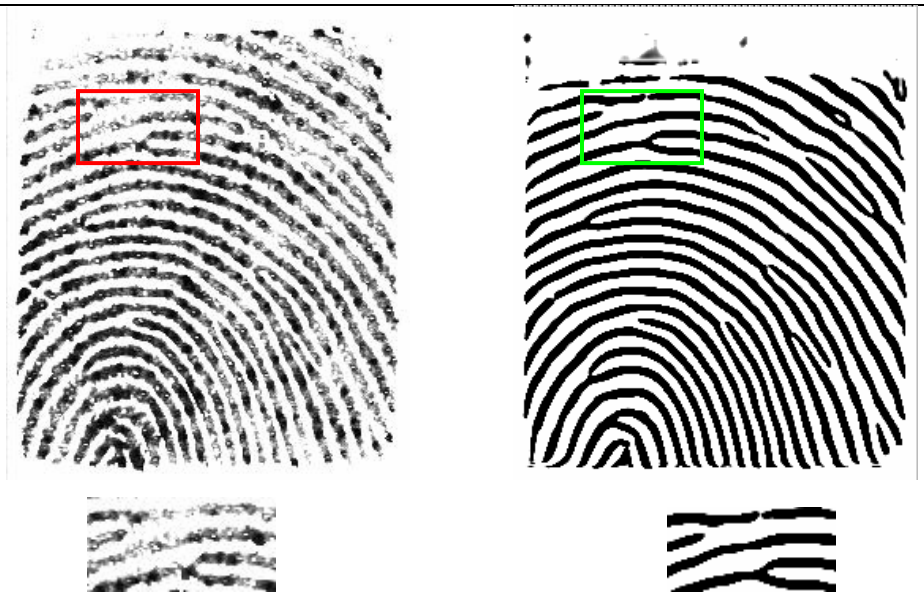


Figure 2.3 Fingerprint image in good quality



a) Original image b) Enhanced image

Figure 2.4 Enhancement effect

After image enhancement, the ridges are thinned into lines with single pixel width by thinning algorithms, base on which minutia extraction algorithm can be applied to extract the minutia details. The whole process is described in the following flow chart:

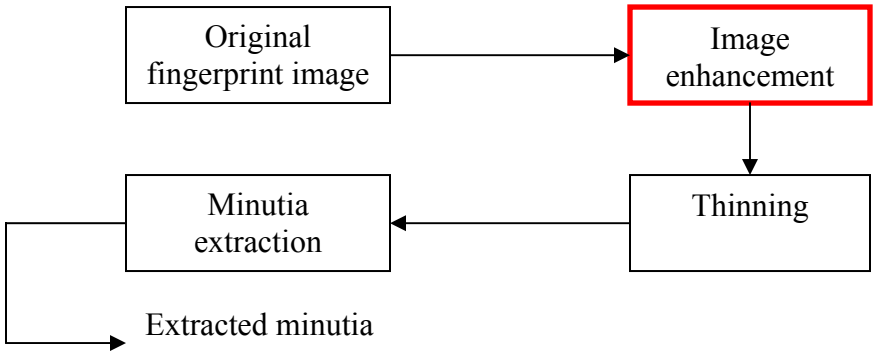


Figure 2.5 Flowchart of the minutiae extraction algorithm

2.2.2 Image enhancement algorithm

The Gabor filter based enhancement algorithm proposed by Hong [1], [2], mainly includes five steps, which is shown in figure 2.6. Based on this algorithm, I decide to add

two more steps, Gaussian filtering and local normalization (the one in figure 2.6 can be considered as global normalization).

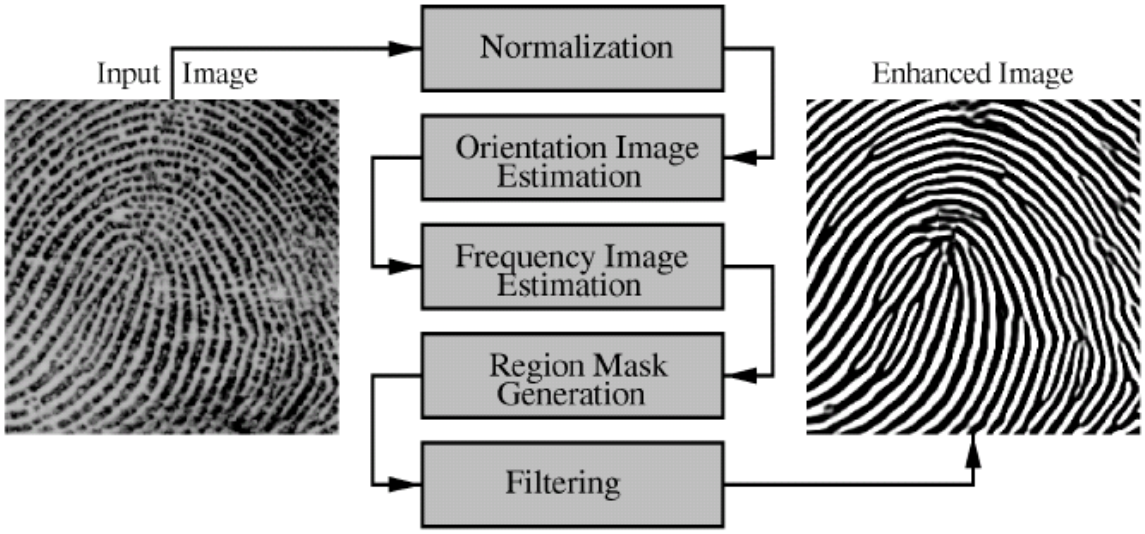


Figure 2.6 Flowchart of enhancement algorithm proposed by Hong [2]

Since we use a capacity scanner as the fingerprint capture device, it's more often that hollows and discontinuities may present in the ridges as showed in figure 2.2. These facts bring great difficulties in finding out the accurate ridge frequencies which is an importance parameter of the Gabor filter. I decide to add two more steps which bring improvement in the enhancement, *Gaussian smooth filter* and *local normalization*. Gaussian smooth filter can cope with most of the hollows and small discontinuities. And the local normalization works as a supplement for the global normalization in case that there are diluted parts in the image.

2.2.2.1 Global normalization

Normalization is carried out so that the input fingerprint has a pre-specified mean and variance.

Let I(i,j) denote the gray-level value at pixel (i,j), M and VAR denote the estimated mean and variance of I, respectively, and G(i,j) denote the normalized gray-level value at pixel (i,j). the normalized image is defined as follows:

$$G(i,j) = \begin{cases} M_0 + \sqrt{\frac{VAR_0 (I(i,j) - M)^2}{VAR}} & \text{if } I(i,j) > M \\ M_0 - \sqrt{\frac{VAR_0 (I(i,j) - M)^2}{VAR}} & \text{otherwise} \end{cases}$$

where M_0 and VAR_0 are the desired mean and variance values, respectively. The main purpose of normalization is to reduce the variance in gray-level value along ridges which will facilitate the subsequent processing steps.

(a) Original image

(b) Global normalization

(c) Local normalization after Gaussian smooth filtering

Figure 2.7 Example of normalization with pre-specified mean = 100 and variance = 100

** Image (b) and (c) are obtained from the software implementation witch will be discussed in next chapter.*

2.2.2.2 Local normalization

When the local mean and variance have large difference to the global one, global normalization usually failed to achieve the goal in those parts. Thus, the normalization is carried out with the local variance and mean values, that is, local normalization. Usually, the fingerprint image is divided into small parts of 16 by 16 pixels and normalized respectively.
Local normalization is applied after Gaussian smooth filtering, as shown in Figure 2.7 (c).

2.2.2.3 Gaussian filter

The Gaussian smoothing operator[5] is a 2-D convolution operator that is used to `blur' images and remove detail and noise. In this sense it is similar to the mean filter, but it uses a different kernel that represents the shape of a Gaussian (`bell-shaped') hump. This kernel has some special properties which are detailed below.

The Gaussian distribution in 1-D has the form:

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

σ is the standard deviation of the distribution. We have also assumed that the distribution has a mean of zero (*i.e.* it is centered about the line x=0).

In 2-D, an isotropic (*i.e.* circularly symmetric) Gaussian has the form:

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

The idea of Gaussian smoothing is to use this 2-D distribution as a `point-spread' function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we

can perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution mask, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the mask at this point. The following table shows a suitable integer valued convolution mask that approximates a Gaussian with an σ of 1.4.

$$\frac{1}{115}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Figure 2.8 Discrete approximation to Gaussian function with σ=1.4

As Gaussian filtering removes the noise at the cost of removing some details at the same time, we may lose some minutiae information by directly applying it. This is especially serious when there are large variances along the ridges. Thus it's necessary to do the Gaussian filtering after global normalization.

(a) Without normalization (b) with normalization

Figure 2.9 Gaussian filtering

** Image (a) and (b) are obtained from the software implementation.*

As in figure 2.9 (a), new gaps are introduced in the ridges, seriously lower the image quality. This is because all the slight details are missing after Gaussian smooth filtering. While in (b), those slight details are strength by global normalization first, and after Gaussian filtering, most of the hollows are scraped off and the ridges are more consecutive. This will help to generate more accurate ridge orientation and frequency.

2.2.2.4 Orientation estimation

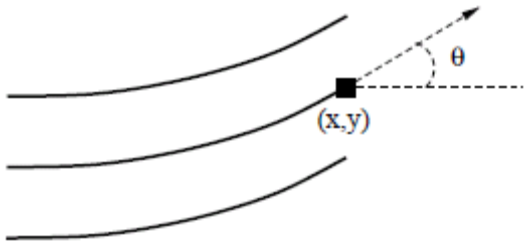


Figure 2.10 The orientation of a ridge pixel

As mentioned before that ridge frequency and orientations are both important parameters in Gabor filter. Hong has proposed an efficient algorithm called Least Mean Square Estimation. In this algorithm, the ridge orientation is computed in block-wise, which theoretically presents the direction orthogonal to the dominating direction of Fourier spectrum of the block. However, I choose to do it pixel-wise instead of block-wise since it will be more accurate when the orientation changes sharply, and usually, it is where the minutiae exits.

Before the orientation estimation, an extra step is needed to compute the gradients for each pixel. There are many methods for spatial gradient measurement depending on different accuracy requirements. Sobel operator[5] is used to find the approximate absolute gradient magnitude at each pixel in an input gray-scale image.

In theory at least, the operator consists of a pair of 3×3 convolution masks as shown below. One mask is simply the other rotated by 90°.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

After applying the convolution masks to every pixel, the gradients in both x and y direction are obtained for orientation estimation. Then a W×W sized block is centered at each pixel (i,j). The orientation can be estimated by following equations, where θ(i; j) is the least square estimate of the local orientation at the block centered at pixel (i; j).

$$
\begin{aligned}
V_x(i,j) &= \sum_{u=i-\frac{W}{2}}^{i+\frac{W}{2}} \sum_{v=j-\frac{W}{2}}^{j+\frac{W}{2}} 2G_x(u,v)G_y(u,v), \\
V_y(i,j) &= \sum_{u=i-\frac{W}{2}}^{i+\frac{W}{2}} \sum_{v=j-\frac{W}{2}}^{j+\frac{W}{2}} (G_x^2(u,v) - G_y^2(u,v)), \\
\theta(i,j) &= \frac{1}{2} tan^{-1}(\frac{V_x(i,j)}{V_y(i,j)}),
\end{aligned}
$$

To reduce the incorrect estimated orientations due to noise presented in the input image, a Gaussian smooth filter is once more needed. First, the orientation image is converted into a continuous vector field, which is defined as follows, where Φ_x and Φ_y are the x and y components of the vector field, respectively.

$$\Phi_x(i,j) = cos(2\theta(i,j)),$$

$$\Phi_y(i,j) = sin(2\theta(i,j)),$$

After the vector field has been computed, Gaussian smoothing is then performed as follows, where G is a Gaussian low-pass filter of size $W_\Phi \times W_\Phi$.

$$\Phi'_x(i,j) = \sum_{u=-\frac{w_\Phi}{2}}^{\frac{w_\Phi}{2}} \sum_{v=-\frac{w_\Phi}{2}}^{\frac{w_\Phi}{2}} G(u,v)\Phi_x(i-uw, j-vw),$$

$$\Phi'_y(i,j) = \sum_{u=-\frac{w_\Phi}{2}}^{\frac{w_\Phi}{2}} \sum_{v=-\frac{w_\Phi}{2}}^{\frac{w_\Phi}{2}} G(u,v)\Phi_y(i-uw, j-vw),$$

Finally, the smoothed orientation is defined as:

$$O(i,j) = \frac{1}{2} tan^{-1} \frac{\Phi'_y(i,j)}{\Phi'_x(i,j)}.$$

Figure 2.11 gives an example of orientation estimation by the LMSE algorithm. The white lines represent the estimated orientation at the center of a 16 by 16 sized window.

Figure 2.11 Orientation estimation

**The two images are obtained from the software implementation.*

2.2.2.5 Frequency estimation

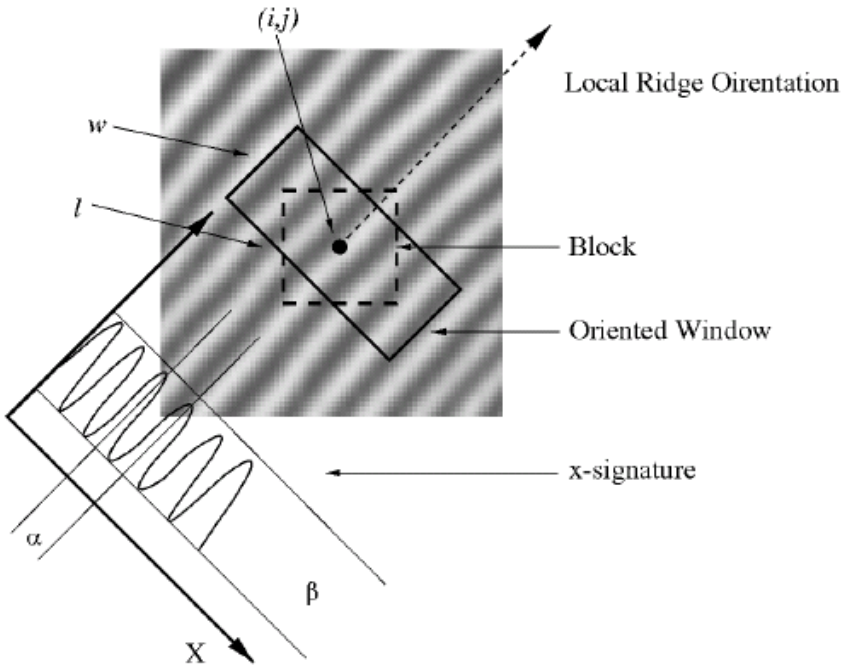


Figure 2.12 Oriented window and x-signature

Besides ridge orientations, ridge frequency is another important parameter in Hong's algorithm. He models the gray levels along ridges and valleys as a sinusoidal-shaped wave along a direction orthogonal to the ridge orientation. With this model, the frequency is computed pixel-wise. But there are some limitations in this model. Firstly, in a real fingerprint image, the ridge frequency is not a constant as in a sinusoidal-shaped wave but may slightly change in space. Secondly, due to the noise, corrupted ridge and valley structures, minutiae, etc, the estimated frequency may be completely incorrect in some pixels. Although with the help of low-pass filter we may reduce these incorrect estimations in a certain extent, but for some images with poor quality it will degrade the enhancement obviously.

For the first limitation, it is relatively trivial because the ridge frequency of a real fingerprint always changes so slightly that we may consider it as a constant. But for the second one, it directly degrades the strength of the algorithm in some case. The first step

of Gaussian smooth filter before enhancement may be considered as an alleviation of the second limitation. However, I still haven't found a complete solution to get rid of the interruption of the presence of hollows and corruptions in the ridges and valleys.

Despite of these drawbacks, I consider Hong's algorithm as the most feasible and efficient one till now. The steps involved in local ridge frequency estimation are as follows:

- As in orientation estimation, the frequency estimation is done pixel-wise. Set an oriented window of size l × w (32 ×16) that is defined in the ridge coordinate system (see figure 2.12) on each pixel.
- For each pixel (i, j), compute the x-signature, X[0], X[1], ... X[l - 1], of the ridges and valleys within the oriented window, where

$$X[k] = \frac{1}{w}\sum_{d=0}^{w-1} G(u, v), \qquad k = 0,\ 1,\ \ldots,\ l-1,$$

$$u = i + \left(d - \frac{w}{2}\right)\cos O(i, j) + \left(k - \frac{l}{2}\right)\sin O(i, j),$$

$$v = j + \left(d - \frac{w}{2}\right)\sin O(i, j) + \left(\frac{l}{2} - k\right)\cos O(i, j).$$

In fact, x-signatures are the average values of the gray-scale values at the same frequency level (with the same local ridge orientation) and step in pixels (the least significant unit in digital image) along the frequency direction to finish all the x-signatures. Thus, the frequency of ridges and valleys can be estimated form x-signature. The average number of pixels between two consecutive peaks can be considered as the period and its reciprocal is the ridge frequency.

- For a fingerprint image captured at a fixed resolution, the value of the frequency of the ridges and valleys in a local neighborhood lies in a certain range. For a 500dpi image, this range is [1/3, 1/25]. Therefore, if the estimated value of the frequency is out of this range, then the frequency must be replaced by the average value of its neighbors.

- The last step is to smooth the frequency by Gaussian smooth filter.

2.2.2.6 Region mask generation

In Hong's algorithm, he didn't give the detail algorithm for the region mask generation because he uses some experience patterns. There are also some other algorithms proposed in [1] and [6]. The algorithm proposed in [6] is quite a powerful one but of course it is quite complicated and may be quite time consuming. Considering that the region mask is used to cut off the empty margin in the fingerprint image and does not effect the enhancement, I decide to choose the simpler one in [1].

The metric used to decide the region of interest is certainty level which is defined as follows,

$$CL(i,j) = \sqrt{\frac{1}{W \times W} \frac{(V_x(i,j)^2 + V_y(i,j)^2)}{V_e(i,j)}}$$

where

$$V_e(i,j) = \sum_{u=i-\frac{W}{2}}^{i+\frac{W}{2}} \sum_{v=j-\frac{W}{2}}^{j+\frac{W}{2}} (G_x^2(u,v) + G_y^2(u,v))$$

For each pixel, if its certainty level of the orientation field is below a certain threshold Ts, then the pixel is marked as a background pixel. Because it's not related to the intrinsic property of fingerprints, it works badly under heavy noise.

The region mask generation algorithm can be incorporated with the orientation estimation since they have similar data flow.

2.2.2.7 Gabor filter

A 2-D even-symmetric Gabor filter performs a low pass filtering along the ridge orientation and a band pass filtering orthogonal to the ridge orientation. Because it's tuned to the two intrinsic properties of fingerprint, ridge orientation and ridge frequency,

it can efficiently remove the undesired noise and preserve the true ridge and valley structures. Gabor filters have both frequency-selective and orientation-selective properties and have optimal joint resolution in both spatial and frequency domains.

The general form of an even-symmetric Gabor filter is showed in follow, where θ is the orientation of the Gabor filter, f is the frequency of the cosine wave, σ_x and σ_y are the standard deviations of the Gaussian envelope along the x and y axes, respectively, and x_θ and y_θ define the x and y axes of the filter coordinate frame, respectively.

$$G(x, y; \theta, f) = \exp\left\{-\frac{1}{2}\left[\frac{x_\theta^2}{\sigma_x^2} + \frac{y_\theta^2}{\sigma_y^2}\right]\right\}\cos(2\pi f x_\theta),$$

$$x_\theta = x\cos\theta + y\sin\theta,$$

$$y_\theta = -x\sin\theta + y\cos\theta,$$

In Hong's algorithm, σ_x and σ_y are chosen as constants and both set as 4.0 according to his experience.

The Gabor filter is generated for each pixel, thus it's a time consuming work. In [7], [8], the author made some modifications to the traditional Gabor filter and found a nice balance between accuracy and complexity. From experimental results, the author found that the effectiveness of Gabor filtering is very sensitive to θ and σ_x , but relatively less sensitive to σ_y. And it is easy to understand because the orientation θ is an intrinsic property of the fingerprint and σ_x influences both the noise smoothing intensity and the contrast between the ridge and valley in the filtering output while σ_y only determines the intensity of the noise smoothing along the local ridge orientation. Thus let $\sigma_x = \sigma_y = \sigma$, the form of Gabor filter becomes

$$G(x, y; \theta, f) = \exp\{-\frac{1}{2\sigma}[x^2 + y^2]\}\cos(2\pi f x_\theta) \qquad (*)$$

Then the Gabor filter is applied to the fingerprint image by spatial convolution with the corresponding ridge orientation and frequency. It's showed in following equation:

$$E(i,j) = \sum_{u=-\frac{w_x}{2}}^{\frac{w_x}{2}} \sum_{v=-\frac{w_y}{2}}^{\frac{w_y}{2}} G(u,v;O(i,j),F(i,j))N(i-u,j-v)$$

In this equation, O is the orientation image, F is the ridge frequency image, N is the pre-processed fingerprint image, and W_x and W_y are the width and height of the Gabor filter mask, respectively.

After a certain σ is chosen, exp $(-1/2\sigma(x^2+y^2))$ can be pre-computed since it's only relative to the pixel's special position in the convolution window. And after the convolution size is chosen to be16 by 16, there are only 256 different values to be pre-computed. The repeat complicated computation for each pixel is replaced by a simple tabulating from the 256 pre-computed values and thus the computational complexity is greatly reduced.

2.3 Conclusion

In this chapter, an efficient fingerprint enhancement algorithm is given based on Hong's work. With this algorithm, FVS can be reliably and widely used for identity verifying and security purpose**.** This algorithm will be implemented in the software implementation in the next chapter and later in the hardware implementation in chapter 4 and chapter 5.

Chapter 3 Software implementation

After choosing and optimizing the algorithms, I wrote a software program in C++ to simulate the algorithms discussed in chapter 2. With a visual user interface, the enhancement result can be examined intuitionally and the time cost is represented by the elapsed instruction cycles of the CPU between the beginning of the first algorithms and the finishing of the last algorithm. In this section, both the enhancement quality and the time cost of the software implementation will be concerned.

This chapter is organized as follows: section 3.1 gives the background of a general software solution for fingerprint enhancement. Section 3.2 presents the software implementation in this thesis. Section 3.3 analyzes the enhancement result from the simulator in both times cost and enhancement quality and the last section draws to a short conclusion.

3.1 Background

As the hardware performance grows greatly in recent years. The software solution for FVS is more and more popular. As shown in figure 3.1, the fingerprint is usually captured by a scanner and then processed by the software program in the PC. This solution is easy to implement by software programming compared with an ASIC solution and benefits from the powerful performance of PC compared with the embedded system platform.

In this chapter, a complete software solution for fingerprint enhancement is given. All the involved algorithms have been discussed in chapter 2 as described in figure 3.2.

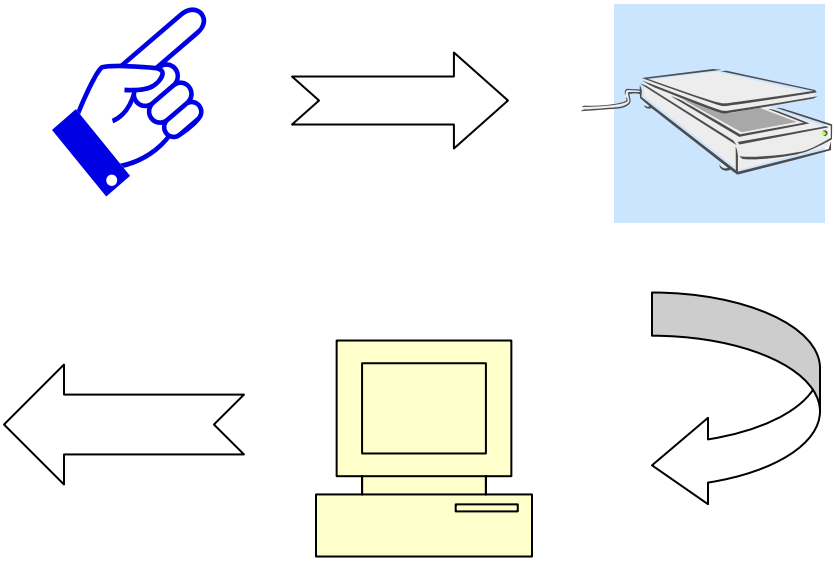


Figure 3.1 Software implementation

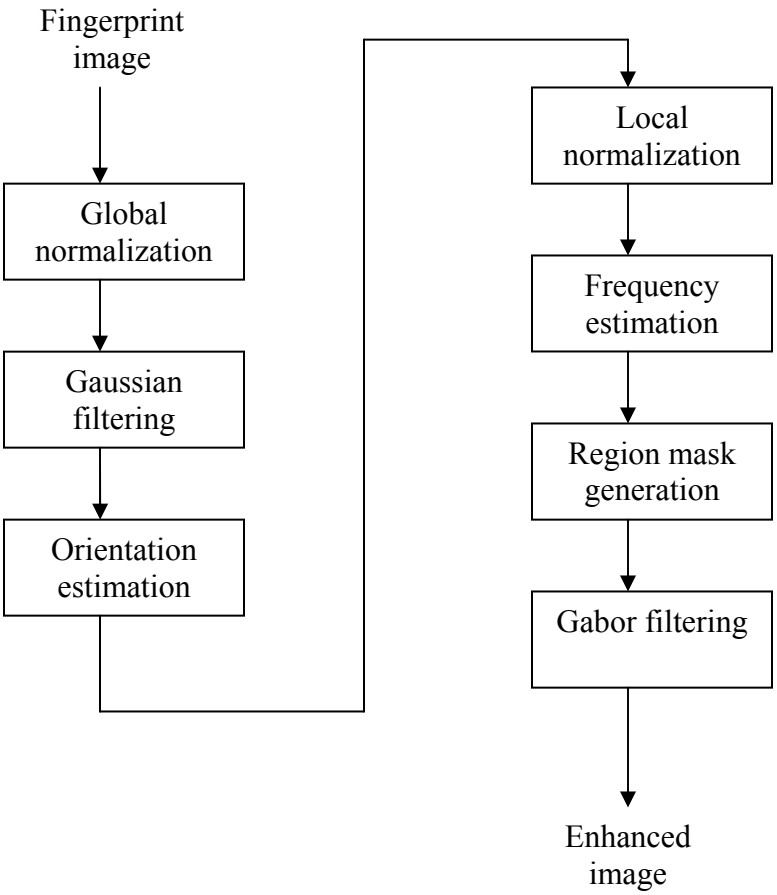


Figure 3.2 Algorithm diagram

3.2 Implementation

Figure 3.3 is a screen shot from the software simulator. All these algorithms are written in C++. The original fingerprint image and the enhanced image are shown on a graphical interface. Each step of the enhancement can be run separately and the intermediate result can be shown on the interface or stored into a binary file respectively.

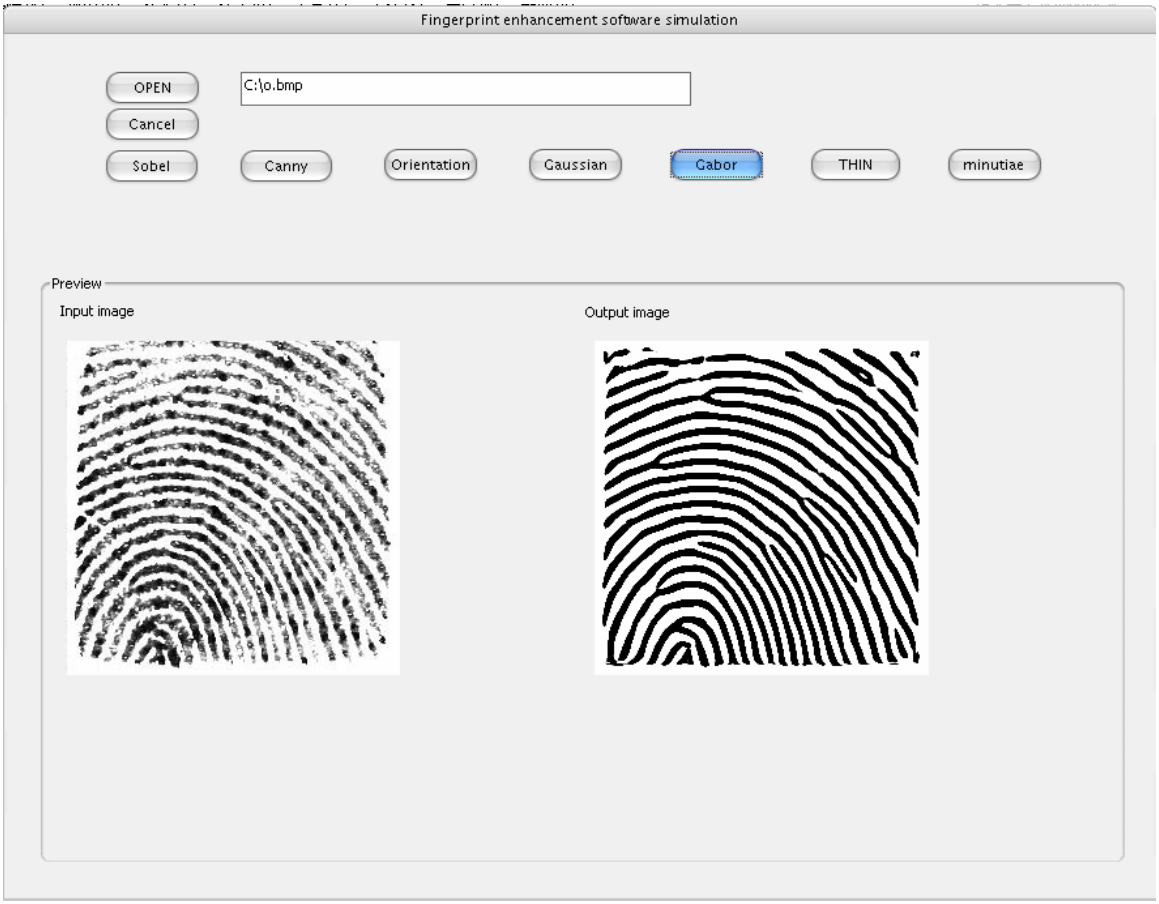


Figure 3.3 Interface of the software implementation

In order to evaluate the time cost of each algorithm due to different parameters and convolution size as well as the optimizations, the simulator incorporates all those possible algorithms discussed in chapter 2 with adjustable parameters. The time cost is calculated by the following formula, where counter and $counter_0$ both are read from Time Stamp Counter (returns the number of clock cycles since the CPU was powered up or reset).

Counter denotes the start time of the algorithm and $counter_0$ denotes the finish time of the algorithm. And 2.2 is the CPU frequency of my laptop (MHz).

$$t = \frac{counter - counter_0}{2.2 \times 10^9} \text{sec}$$

3.3 Simulation result analysis

3.3.1 Time cost and optimization

As stated in chapter 2, the enhancement involves a lot of convolutions which can be quite computational expensive. Thus, the convolution size directly decides the time cost. Further, the orientation estimation, frequency estimation and Gabor filtering also require a lot of trigonometric function calculations which also can be quite time consuming. Besides, we also benefit from the algorithm optimization described in chapter 2.2.2.7. In the following, we will examine how these three facts works on the time cost.

(a) without optimization (b) after optimization

Figure 3.4 Enhancement with different modified Gabor filters in chapter 2.2.2.7

Original (sec)	Modified (sec)	improvement
15.287	9.15	40.1%

Table 3.1Time gain by modified Gabor filter

In figure 3.4, we can see the enhancement is almost the same after we use the modified Gabor filter. However, the time cost is reduced by 6.137 seconds since the original algorithm cost totally 15.287 seconds and after the optimization, it costs 9.15 seconds. Here, the convolution size is chosen as the standard size proposed by Hong.

Yet, 12.276 seconds is still too much for real-time application. On the other hand, by reducing the convolution size, the computational complexity will be greatly reduced. The following table shows the time costs of different convolution size.

#	Convolution size in frequency estimation	Convolution size in Gabor	Time cost (second)	Enhancement quality
i	33	17	9.15	Acceptable
ii	25	13	5.4892	Unacceptable
iii	17	9	2.999	Unacceptable

Table 3.2 Time cost of different convolution size

Obviously, in (ii) and (iii) the time gain is attractive since 3.6608 and 6.151 seconds are reduced in each simplification respectively. However, the enhancement quality is also degraded greatly. Too many discontinuities in the ridges could lead to many spurious minutiae as shown in Figure 3.5, where (a) is the enhancement result of simplification (ii) and (b) of (iii). This is an unacceptable tradeoff between time cost and enhancement quality.

(a) (b)

Figure 3.5 Enhancement results of other convolution sizes

To avoid the overload in trigonometric function calculations, tabulating in a lookup table is an efficient solution in software implementation. Considering that the orientation estimation costs relatively trivial time compared with the frequency estimation and Gabor filtering and the asymmetry exits in tangent function, only the sine/cosine and exp functions are implemented by tabulating. Form the following table, we can see, the total time cost is reduced by 3.112 to 6.038 seconds, an improvement of 34%. And figure 3.6 implies that the enhancement is almost the same and no spurious minutiae will be introduced because of the accuracy losing in lookup tables.

Time cost without tabulating (sec)	**Time cost with tabulating (sec)**	**Improvement**
9.15	6.098	33.36%

Table 3.3 Time gain by tabulating technique

Therefore, with the modified Gabor filter and tabulating technique for trigonometric function calculations, the time cost is reduced by 60.11% without degrading the enhancement reliability. And the total time cost for fingerprint enhancement is reduced to 6.098 seconds. Slightly difference may exists between different input fingerprint images due to different region masks (different backgrounds), but generally, the time cost is not less than 6 seconds.

Figure 3.6 Enhancement with tabulating technique

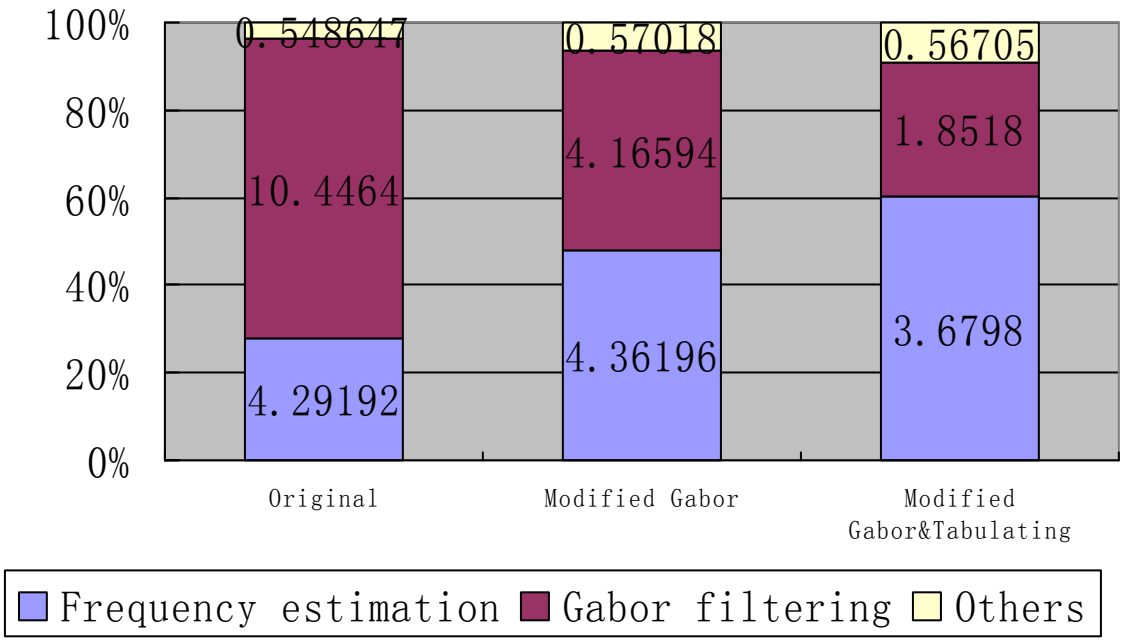


Figure 3.7 Time cost analysis in software implementation

In figure 3.7, it's clear that the frequency estimation and Gabor filtering cost most of the time in fingerprint enhancement. The modified Gabor filter greatly reduces the computational expense in Gabor filtering. And the tabulating technique brings benefits in both frequency estimation and Gabor filtering. This is understandable since both of them require a lot of trigonometric function calculations.

3.3.2 Enhancement quality and algorithm adjustment

There is not a rigid criterion on the quality of the enhancement, but usually, we judge it by our eyes and comparing with others images. On the other hand, the enhancement is an important pre-processing for the minutiae extraction later. In this case, the best enhancement algorithm should restore missing or damaged minutiae information while never or rarely introduce spurious information. However this part will not be discussed in this thesis.

In the simulation results shown in the following figures, it's convictive that not only the convolution size work on the enhancement quality heavily but also the Gaussian filtering and local normalization do have much contributions to the final result. The convolution size describes the spatial relationship between each pixel and its neighbors. The larger the size is, the accurate the spatial information is collected. However, spatial convolution is quite time consuming, and it's proved that for a normal fingerprint image, a 16 by 16 convolution size is the best trade off between accuracy and computational complexity. Gaussian smooth filtering and local normalization are two extra processing steps based on Hong's algorithm. They are efficient to remove the hollows in the fingerprint image and smooth the variances as a supplement to the global normalization in case that there are large contrasts in a fingerprint image.

In figure 3.8, image (a) is the original fingerprint obtained from a capacity scanner; Image (b) is the enhancement result without Gaussian smoothing filtering. It's obvious that the hollows presented in the ridges make the result unacceptable. Image (c) shows how the hollows are eliminated by Gaussian smoothing filter. The large gaps in the ridges result from the discontinuities or large variances of the grayscale value between the ridge pixels. With the help of local normalization, it's greatly released as showed in image (d) . The presence of stains around the fingerprint is due to the region masking algorithm, as mentioned before, a not so powerful but time cost efficient algorithm. These stains can be easily ignored by the extracting algorithm.

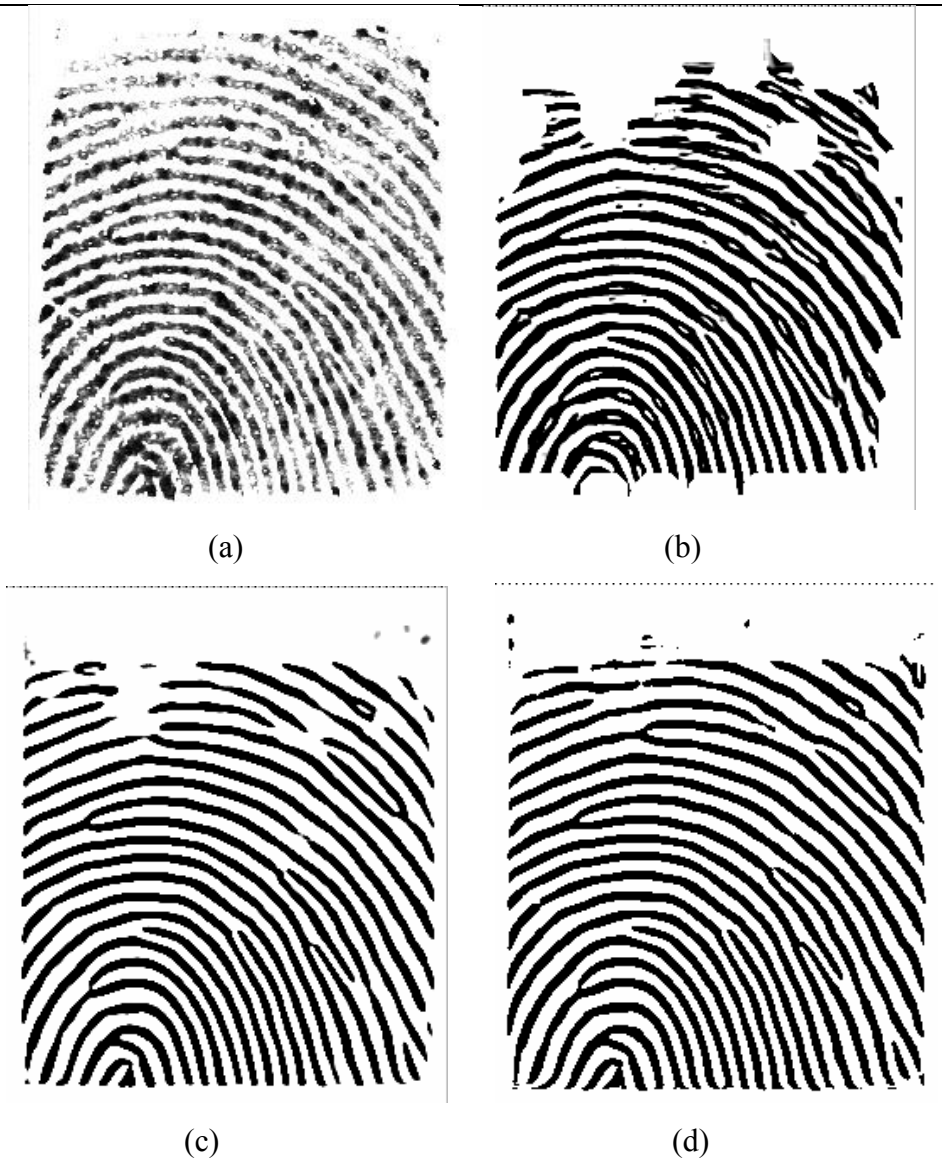


Figure 3.8 Enhancement quality analysis

As another factor, which also contributes to the enhancement result, the convolution size, it doesn't concern about the intrinsic characteristic of the algorithm. The best convolution size is chosen depending on the trade off between the time cost and enhancement quality and there are also differences between input fingerprint images, such as the interval distance between the ridges and valleys. This is easy to understand because the orientation and frequency are obtained by assuming that they are the mean values of all the pixels in a convolution window, and thus, if there are sharp changes in orientation or frequency somewhere, large convolution window will of course get a coarse estimation.

At the last of this section, the minutiae extracted by the software simulator are also given in figure 3.9. Although, we won't discuss the minutiae extraction and thinning algorithm[3] further, but importantly it can tell us how the spurious minutiae is introduced to the extracting algorithm and how a minutiae is restored from the original image. Most of the false minutiae extracted from the enhanced image are caused by unpredictable break off in the ridges and incorrect joint of ridges. These two properties will help us to improve our enhancement algorithm as well as using as an intuitional criterion.

(a) (b)

(c)

Figure 3.9 Enhancement and minutiae extraction results from the simulator (a) Original image. (b) Enhancement result. (c) Minutiae extraction result.

3.4 Conclusion

In this chapter a software solution on fingerprint enhancement is given. After optimization the time cost is reduced to around 6 seconds on a Pentium4 2.2 MHz CPU. However, this is still too large for a real-time application. Thus, efficient hardware solution with acceptable cost is quite urgent and in the rest of this thesis, a fast and low cost SIMD architecture is introduced to meet this goal.

Chapter 4 Architecture study

Compared with the software implementation on a GPU, an application specific hardware solution benefits in both speed and hardware cost, however, this needs elaborated design work.

In the enhancement algorithm, there exits strong independence between each pixel, which means it can potentially benefits from a parallel architecture. As in general image processing, the data transfer is always a heavy load and usually it becomes the bottleneck in system performance due to narrow memory bandwidth. A pre-fetching technique can be used to release this downgrades. Floating-point representation can also be replaced by fix-point representation in order to avoid the FPU.

This chapter is organized as follows: section 4.1 proposes a parallel architecture by exploiting the parallelism exists in the enhancement algorithm. Section 4.2 is the architecture design phase and section 4.3 mainly focuses on the fix-point representation. In the last section, a short conclusion is given.

4.1 Parallelism analysis

Till now, Hong's algorithm is considered as quite an efficient and feasible enhancement algorithm. However, the huge computational complexity due to large numbers of spatial convolution limits its application in embedded and real-time domain with uniprocessor architecture. In general, parallel architecture plays an important role in high speed computing. This section will examine the feasibility in parallel architecture for the algorithms described in the last section.

4.1.1 Data dependence

Not all the algorithms can be processed in parallel. That's because there exits data dependence. Data dependence can largely destroy the parallelism by making the data path overlapping.

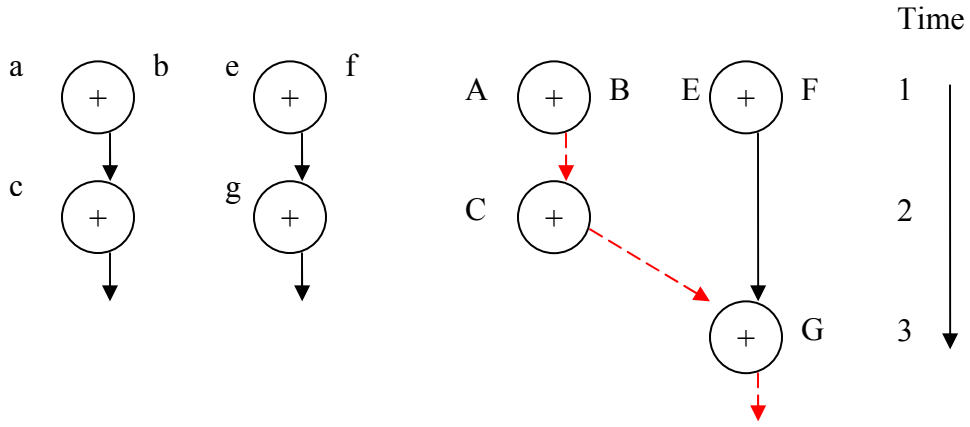


Figure 4.1Data dependence

As shown in figure 4.1, in the left side, with two adders the results can be generated within two time units, comparing with 4 time units for one adder in a sequential solution, and in this case two adder works independently with no data communication. But it's not the same case at the right side. The output of one adder serves as the input of another adder. That's to say, the second adder must wait for the result coming from the first one no matter whether it has been ready for the addition before time 3. The time cost should be that of the longest data path (critical path, noted by dotted arrow), which is 3 in this case.

As in the right part, not all the additions are carried in parallel, but partially. In another word, there is an adder in idle some time. The hardware is not sufficiently used. In some extreme case, there is only one data path, where all the operations must be processed in sequence. Then, a parallel architecture won't bring any speed benefits. Finding out the

dependence in the algorithm and utilizing the intrinsic parallelism can make us greatly benefit from parallel architecture.

The following table shows the data dependence/parallelism in each algorithm that has been chosen for fingerprint enhancement. Note that since there is rarely a full parallel algorithm in which every operation can be carried out in parallel with no data dependence, the least significant unit described in the third column is a piece of sequential operations. And the data needed in these algorithms is assumed to be ready by hand despite the way to get from memory, which will be the topic in next section.

Algorithm	**Sub algorithm**	**Data dependence/parallelism**
Global normalization	Global mean	Sequential, infix sum
	Global variance	Sequential, infix sum
	Normalization	Depending on global mean and variance, Parallel in pixel wise
Local normalization	Local mean	Sequential, infix sum
	Local variance	Sequential, infix sum
	Normalization	Depending on local mean and variance, Parallel in pixel wise

Gaussian smooth filtering	Gaussian smooth filtering	Depending on global normalization, Parallel in pixel wise, involving spatial convolution of size 3 by 3 with constant convolution coefficients
Orientation estimation and region mask generation	Gradient estimation	Parallel in pixel wise, involving spatial convolution of size 5 by 5 with constant convolution coefficients
	LMSE	Depending on gradient estimation and above, Parallel in pixel wise, involving accumulation of large number of products.
	Region mask generation	Depending on gradient estimation and above, Parallel in pixel wise, involving accumulation of large number of products.
Frequency estimation	Frequency estimation	Depending on all above, Parallel in pixel wise, involving accumulation of large number of products.
Gabor filtering	Gabor filtering	Depending on all above, Parallel in pixel wise, involving spatial convolution of size 16 by 16 with variant convolution coefficients

Table 4.1Parallelism analysis of enhancement algorithm

In table 4.1, it's interesting to find out two factors that inspire us to come to a parallel solution.

- **For most of the algorithms (involving most of the computing), it's fully parallel in pixel wise.**
- **These algorithms mainly consist of spatial convolution or accumulation of large numbers of products.**

The first factor tells us that most algorithms apply on each pixel repeatedly with independent data flows. Thus, with n processors working on n pixels simultaneously, it can be n time faster. The maximum value of n is the number of pixels in the fingerprint image.

Factor 2 brings us more imaginations. Besides in mean and variance estimations, both spatial convolution and accumulation of products in other algorithms can be considered as infix sum.

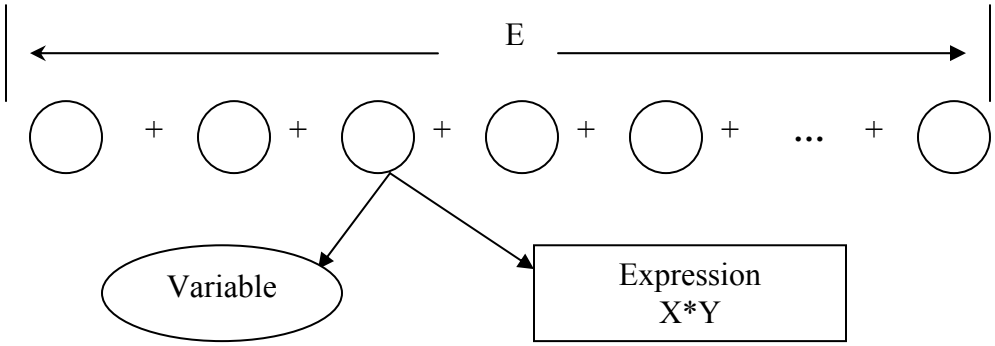


Figure 4.2 Infix sum

The operands in the infix sum can be both variables like in mean or variance estimation, and an expression of multiplication as in Gaussian filtering and frequency estimation, etc. The products can be pre-computed in full parallel with enough number of processors, noted as E. Treated as a binary balance tree (figure 4.3), the infix sum can be finished in $\log_2 E$ time with E/2 processors. Thus totally E processors are needed and the time cost is reduce to $1+\log_2 E$ from 2E.

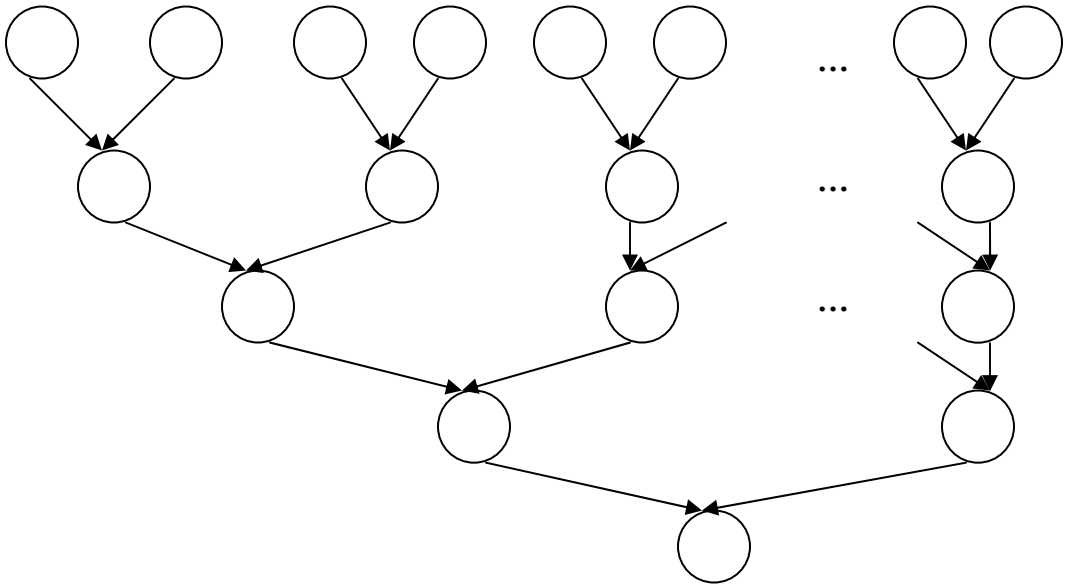


Figure 4.3 Binary balanced tree

4.1.2 Parallel architecture

As discussed in the last section, the fingerprint enhancement algorithm is quite suitable for a parallel architecture. However, considering the hardware cost and complexity in the architecture, not all the time benefits obtained in parallelism are feasible or worth in practice.

Three possible solutions come out based on the analysis in last section, each bringing a considerable speed up compared with uniprocessor solution.

- **Pixel parallel** A processor array of size n is assigned to n pixels, each working for a pixel simultaneously. While in a uniprocessor solution, algorithms are carried out repeated on each pixel, with a processor array, processing be carried out on n pixels in parallel, resulting in an n times speedup. Since each processor works on the same algorithm, there is no processor idle during all the time.
- **Binary balanced tree** E processors can reduce the time from 2E to $1+\log_2 E$ for all the infix sum of products. However, each processor works on different instructions since we split an algorithm into E pieces unequally. For infix sum of variables, E/2 processors are in idle, because they don't need to pre-compute the products. Unfortunately, the infix sum in different algorithms usually has a different

size. For those with a smaller size than E, assumed as S, E-S processors are wasted. And for other sequential algorithms, which can not be divided into several parts in parallel, there will be only one processor used.

- **Combination of both above** In this way, maximum time gain is obtained and of course, it has the highest hardware cost and complexity. The total number of need processor is n*E, really a huge cost.

The speedup of first solution is n times and for the second is less than $2E / (1+ \log_2 E)$ and similarly for the third it's less than $n*(2E / (1+ \log_2 E))$. The reason why speedup can not reach $2E / (1+ \log_2 E)$ or $n*(2E / (1+ \log_2 E))$ for the second and third solution lies in that only infix sum can benefit from the balanced binary tree architecture.

If T and N stand for time gain and number of processors respectively, the efficiency can be represented by the following formula:

$$E = \frac{T}{N} \begin{cases} \approx \frac{N}{N} = 1 & \text{(The first solution)} \\ < \frac{(2E/(1+\log 2\,E))}{E} = 2 / (1+ \log_2 E) & \text{(The second solution)} \\ < \frac{N*(2E/(1+\log 2\,E))}{N*E} = 2 / (1+ \log_2 E) & \text{(The third solution)} \end{cases}$$

**Note that the time gain here just concerns with the processing time, ignoring the time cost in data communication, which can be quite large, especially in image processing.*

Only the first solution has an efficiency of almost 1 (the mean and variance estimation can not be carried fully in pixel parallel) and the rest may be quite low when sequential algorithms take a large ratio or E is too large, depending on different algorithms. Since our purpose is aiming on mobile embedded devices, the constraint on hardware cost and

complexity can be crucial. Pixel parallel seems to be a feasible and efficient solution for us.

The reason why pixel parallel solution has such a high efficiency is that most of the algorithms are in parallel in pixel wise. That means every processor has the same instructions (data flow) and works on different data streams. This is a typical SIMD (single instruction stream, multiple data stream) architecture [9], [12]. The basic architecture is shown in figure 4.4.

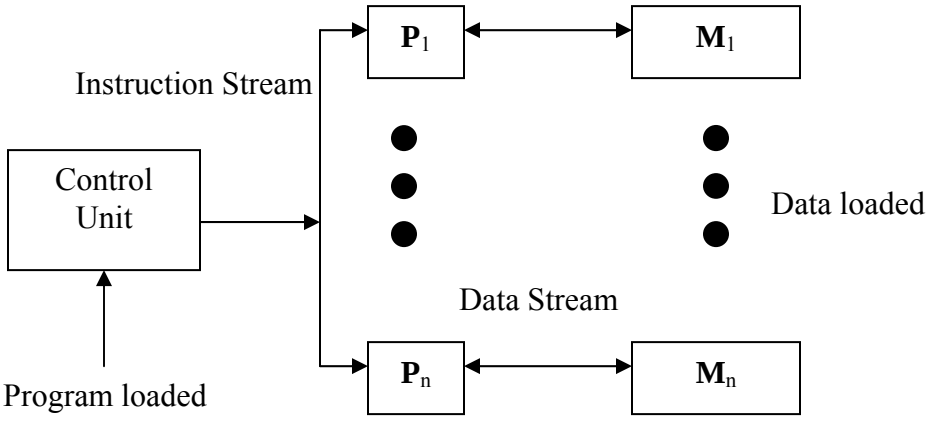


Figure 4.4 SIMD architecture

4.2 Architecture design

In [10], the author proposed a similar SIMD architecture called SIMD Pixel Processor (SIMPil) architecture and it's proved to be successful in greatly speeding up many basic image processing algorithms. However, there are three main differences compared with in this thesis. (a) Their prototype of SIMPil processor was designed in 0.8 μ m CMOS process and fabricated through MOSIS. There is much less constraint in hardware area while we originally aim to implement this architecture on Xilinx Spartan XC3S2000 chip, and hope this architecture can be applied on many FPGA chip with acceptable price. (b) In their design, each processor directly fetches image data from sensor simultaneously, avoiding the system bottleneck caused by limited memory bandwidth. In this thesis,

external memory is the only way to get image data and thus inevitably limited by the memory bandwidth. (c) In this thesis, we hope the design could not only work on basic image processing algorithms such as special convolution but also can deal with common sequential algorithms and in general, implement many complicated image processing algorithms (like fingerprint enhancement) on a smart card.

4.2.1 Parallel processor array

Based on the hardware constraint of Spartan XC3S2000 chip [11], the size of the processor array is limited to 4 by 4 in the implementation of this thesis. That's mainly because of the limited on chip memory space. However, the architecture should be scalable and can be resized according to different requirements to achieve the desired performance.

The size of the array ranges from 1 to N, where N is the number of pixels in the fingerprint image. When choosing 1 as the size of the processor array, it is more or less the same as traditional software implementation. The difference is that the software implementation uses a general purpose processor, while in our case we can design a dedicated processing unit that focuses on the operations involved in enhancement algorithms.

There are 40 dedicated multipliers on Spartan 3 xC3S2000 chip. With these multipliers, we can implement the convolution operations at a low hardware cost with a considerable high performance. Thus, the size of processor array can be as large as 40. Making the size of processor array to be power of 2 will greatly convenient our algorithm implementation and decreasing the hardware cost (we can always use a shifter instead of a multiplier when the multiple is power of 2). Therefore, 16 or 32 could be a suitable size of the processor array. As the number of processor increases, the required memory and cache size also increases. However, there are only maximally 320k bits distributed RAM and 720K bits block RAM. The size of processor array is limited to be 4 by 4.

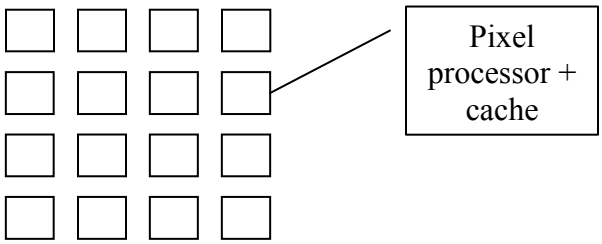


Figure 4.5 Processor array

Suppose the pixel processors are assigned to the pixels with the same spatial shape. As a matter of fact, there are many other possible spatial shapes for the processors array as shown in the following figure.

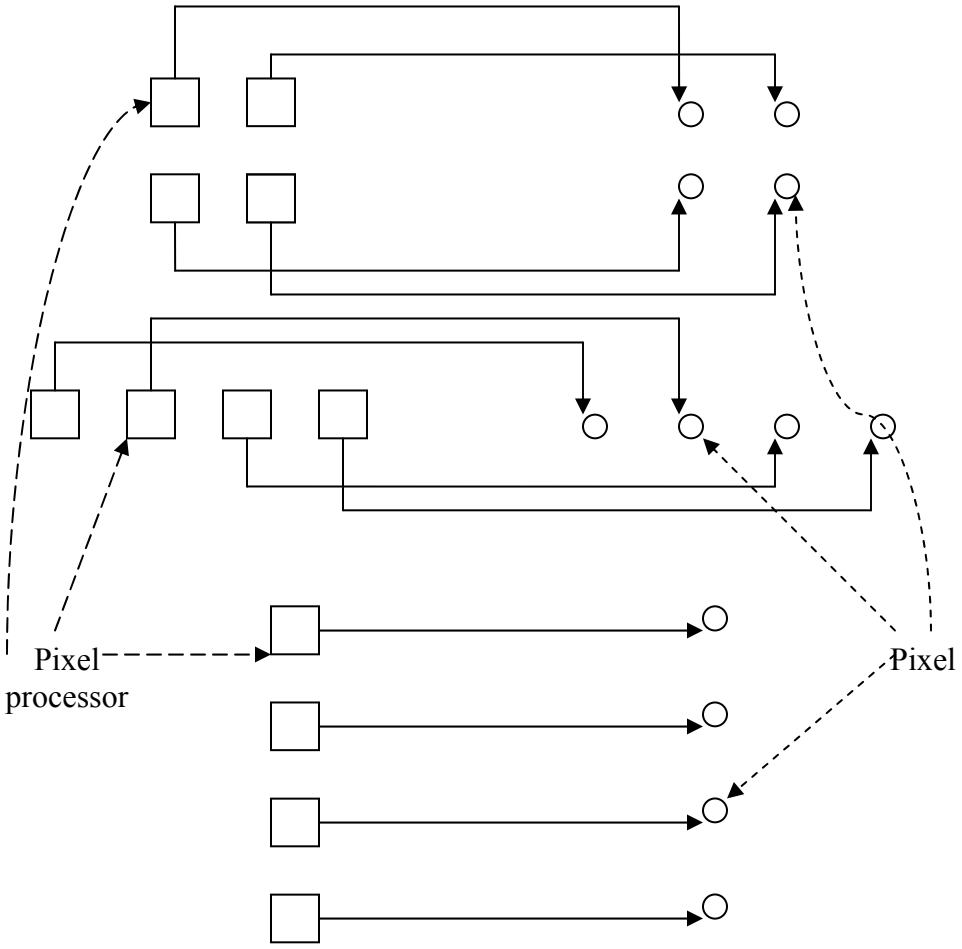


Figure 4.6 Spatial shapes for pixel processor array and its mapping relationship with pixels

There are two major reasons that make the shape of square better others.

- In image processing, most of the data dependences or algorithm level relationships exist in the neighbor pixels. A square shape covers more such relationships than others. This can increase the adaptability of the SIMD architecture for different algorithms.
- A square shaped processor array can avoid more redundant external data reading than others. This will be discussed later in this chapter.

Thus, the size of processor array in this thesis is chosen to be a 4 by 4 and distributed in a square shape.

4.2.2 Multiprocessor interconnection

In the algorithm analysis, as mentioned before, all the algorithms are independent in pixel wise except mean and variance estimation. And usually in image processing, most operations can be carried out with out data dependence on others pixels. In some case, it's required to collect or process the information of the whole image or locally a part. If there is no communications between processors, the processing is forced to be sequential as in uniprocessor architecture.

On the other hand, since the image data is fetched from external memory, there is no need to pass image data between neighbor pixels. Only in those algorithms where data dependence exists between pixels, communication between pixels is necessary to avoid processing in full sequential.

There are two kinds of communications between image pixels, short communication (between neighbors) and long communication (to a remote pixel). Short communication usually involves exchanging data with its four neighbors as shown by arrows in the right of figure 4.7, or sometimes, with eight neighbors, including the doted arrows. Long communication needs a more elaborated communication mechanism and costs more time, since data pass through more than one pixels and need path control. Fortunately, there are rarely requirements for long communication in image processing. And in case there really

is a need, external memory can be used as a relay. Considering the complexity, in this thesis, we only discuss the short communication.

Despite the long communication and the corresponding complicated interconnection network, eight connections for each processor to its neighbors still seem to be redundant to an application specified design. In fingerprint enhancement, only mean and variance estimations require interacting communications, which require accumulating the mean or variance values of each pixel. By dividing and assigning the work to each processor, the algorithm can be processed as two steps. Firstly each processor accumulates a number of

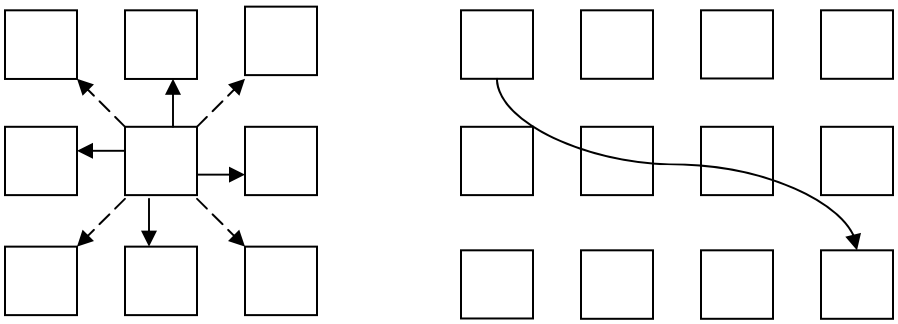


Figure 4.7 Data communication between pixels

values in sequential and then sum up the results from each processor. The second step is a kind of infix sum as discussed in chapter 2.2. Considering each processor as an element in the binary tree, a connection as in Figure 4.8 can finish the sum in log n time.

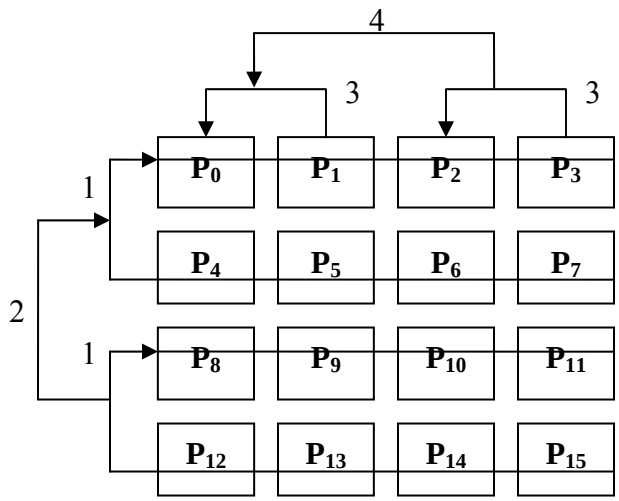


Figure 4.8 Interconnection of the processor array

Note that some processors have more than one connections to its neighbors and the rest have only one. A simple dynamic interconnection network can reach the goal and makes every processor have the same structure.

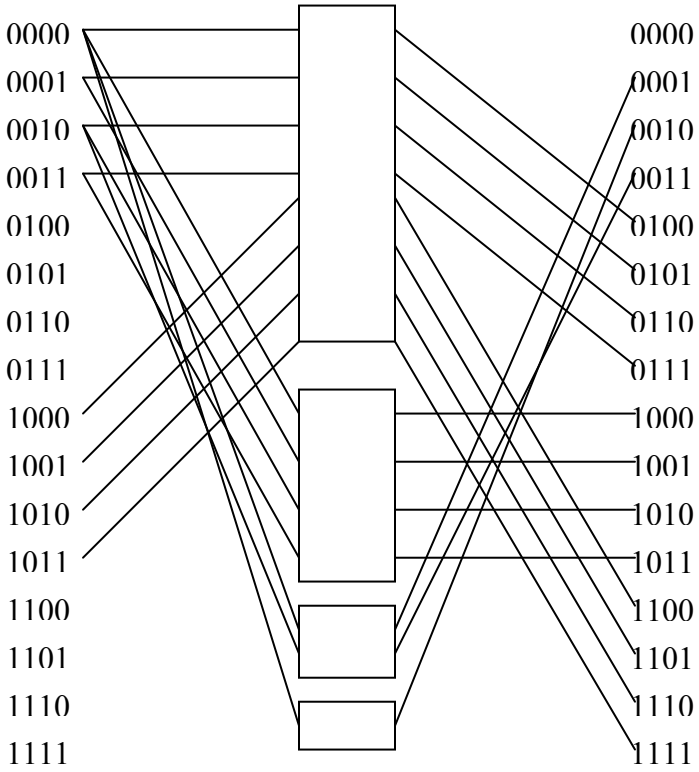


Figure 4.9 A single stage dynamic interconnection network

4.2.3 Memory architecture

There are two classical memory architectures for parallel architecture, shared memory architecture and message passing architecture. A shared memory system typically accomplishes inter-processor coordination through a global memory shared by all processors. And a message passing system (also referred as distributed memory) typically combines the local memory and processor at each node of the interconnection network. For a multiprocessor with a shared memory, the shared memory address space can be used to communicate data implicitly via load and store operations while for a multiprocessor with multiple address spaces (distributed memory), communication of data is done by explicitly passing messages among the processors.

As discussed in last section, in fingerprint enhancement algorithms, all the required communications among processors can be accomplished through a simple dynamic interconnection network in the processor array. Thus it has nothing to do with the memory architecture. The memory architecture should be efficient only in the data load and store, which can be quit a heavy load in image processing.

After capturing the fingerprint from a capacity sensor, the fingerprint image is stored in the external memory. All the processor must load the image data from external memory. And thus, it seems a shared memory architecture is suitable in this case. Since there is only one 32bit bus connecting the external memory and the multiprocessor system, it turns out to be a single bus shared memory architecture (figure 4.10).

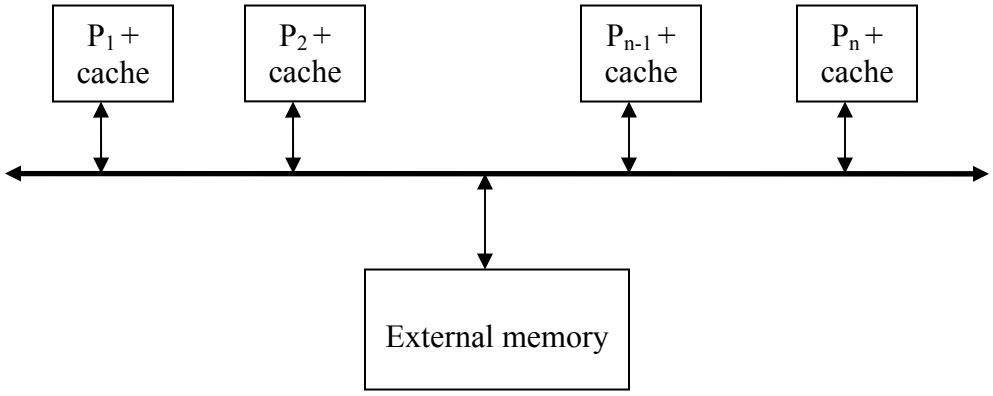


Figure 4.10 Single bus shared memory architecture

This architecture is simple to implement and the use of local caches reduces the processor-memory traffic. However, it's inherently limited by the bandwidth of the bus and the fact that only one processor can access the bus, and in turn only one memory access can take place at any given time. Especially, for a SIMD system, this can greatly degrade the performance of the whole system, since each processor works under the same instruction streams and they always load and store data synchronously. Another problem that can lead to the degradation is coherence problem. Having multiple copies of data, spread throughout the caches can alleviate the limitation of memory bandwidth. However, it might also lead to a coherence problem. The copies in the caches are coherent if they are all equal to the same value. But if one of the processors writes over the value of one of the copies, the copy becomes inconsistent because it no longer equals the value of the other copies. Although a variety of solutions can deal with this problem, it may largely increase the complexity of the system and still the degradation of performance is inevitable.

If every processor has its own memory space, there won't be a bottleneck in memory bandwidth, but still, we may face the problem that there is only one data bus connecting to the external memory where the fingerprint image originally exits. What' more, due to the hardware area constraint, we can not have a big enough distributed memory space for each processor to hold all the data it needs during the whole processing.

Frequent data load and store between multiprocessor and external memory is inevitable. That means lots of time will be cost on transferring data through the 32bit wide bus. Now that it is inevitable, there is only one way to get rid of it – hiding it or overlapping it with computation or other communication.

In traditional way, the processor works in a manner like: load – process – store. When it is in process stage, the bus is in idle. If it's possible to load the data for next process stage during the processing time, the data loading is hidden behind the computation!

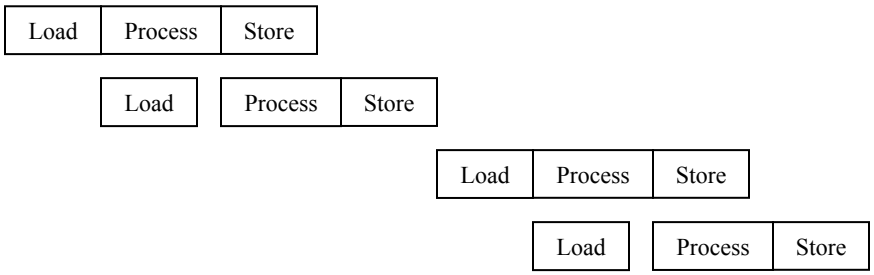


Figure 4.11Overlapping load stage and process stage

The store stage can not be overlapped with load stage because there will be a conflict in using a single bus. And two limitations make it impossible to store the result into local cache: (a) we may face a coherence problem; (b) there will not be enough space to hold the results in cache until the finish of enhancement. Fortunately, the store stage usually costs a trivial time compared with the first two stages.

To achieve the goal of overlapping load stage and process stage as shown in figure 4.11, two conditions must be met:

- Double the local memory size. Actually, it should be called local cache. One piece of cache holds the data for the process stage and another gets the new data for next process.
- Predict the data required in next process stage.

The first condition is easy to achieve but the second one usually is not feasible in practice. Once there is a data dependence between two consecutive process stages, the coming

load stage can only begin when the new data is ready right after the last process stage finishes.

In this pixel parallel SIMD architecture, as discussed before, every processor is assigned to a pixel at a time and then to another. Since there is no data dependence among the data flow of each pixel, it's free to predict the input data for the next process stage. As to the variance and mean estimations, the input data can also be divided into small independent parts although the processing can not be fully in parallel as it is carried out like a binary balanced tree. Thus, the data loading and processing can be fully overlapped with each other by pre-fetching the data for next process stage. Usually the time costs for the two stages are not identical, and it's necessary to wait for the longer one in order to synchronize them. Let L, P and S stand for the time cost of the three stages respectively, and then the time gain due to the pre-fetching can be expressed by the following formula:

$$\text{Time gain} = \frac{L + P + S}{\max(L, P) + S}$$

When L=P, the time gain reaches its peak.

With a control unit, called SIMD controller, the same instruction stream is sent to each processor as well as the data streams are pre-fetched and sent to each processor. The controller loads and dispatches the data for all the processors altogether instead of managing by the each processor respectively. There are two advantages to do so. First, it avoids a mechanism in solving the conflict in using a single bus simultaneously. And we may benefit in time cost from another advantage. If some of the processors require the same data in the external memory, the controller will load the data once and dispatch to each processor simultaneously instead of loading several times respectively for each processor. This situation can be quite often in the image processing, since the required data for one pixel are usually from an area around it. When the processor array assigned to a block of consecutive pixels, the required input data is strongly overlapped! An example is given in figure 4.12, if the input data for one pixel lies in a 3 by 3 block around it, a 4 by 4 processor array need $16 \times 9 = 144$ values. But most of them are

overlapped, and in fact only 6 × 6 = 36 different values need to be loaded from the external memory. The efficiency will be boosted by 144 / 36 = 4 times!

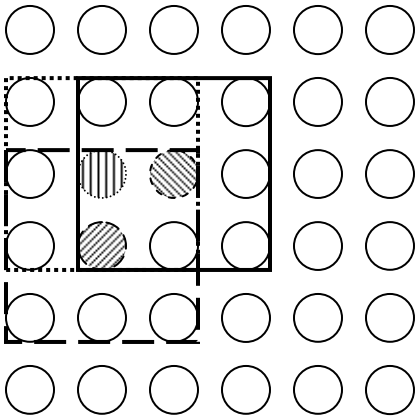


Figure 4.12 Overlapping of input data for the processor array

After the SIMD controller dispatching data to every processor, there will be a piece of local cache to hold these input data till the finish of next process stage. Considering for some algorithms, it can be larger than 2k bytes and totally 32 bytes for the 4 by 4 processor array, the local cache is chosen to be built up with dual port synchronous block RAM. That means a data fetch from these block RAM will cost an extra cycle. To speed up the data fetch and also store the intermediate results in the process stage, we also need a number of distributed caches/registers in the pixel processor. To distinguishing these two caches, we call the block RAM cache as pixel memory and the distributed cache as pixel processor cache. Figure 4.13 gives a global architecture of the SIMD system.

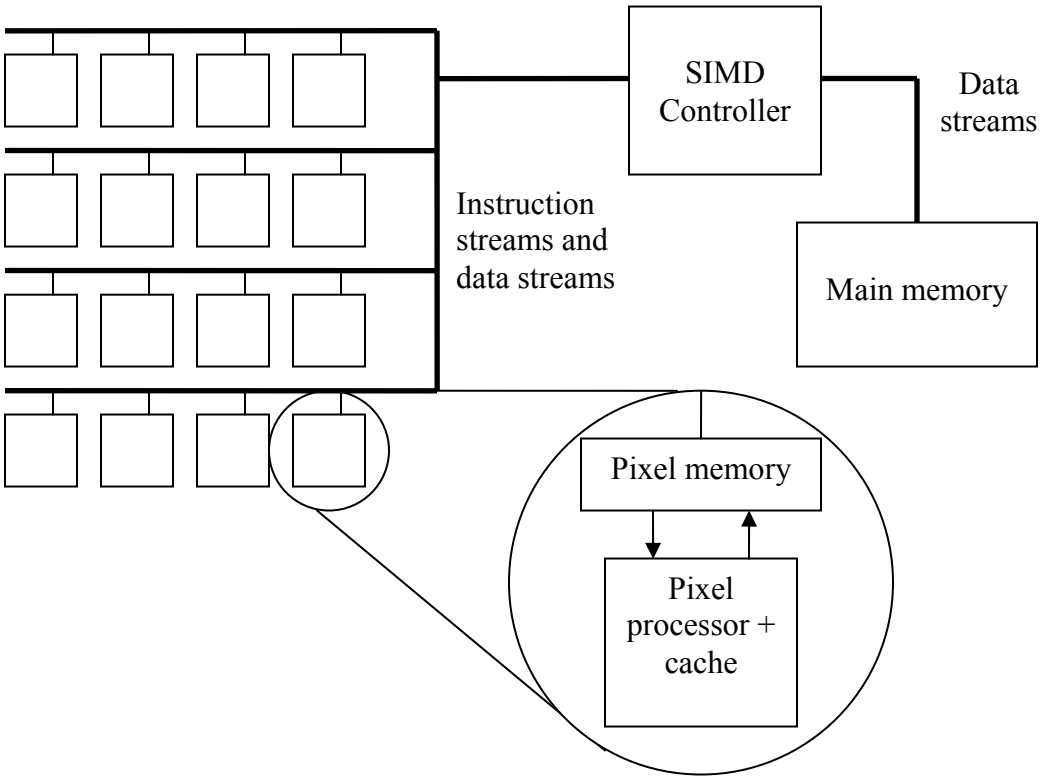


Figure 4.13 Memory architecture

4.2.4 Design of pixel processor

4.2.4.1 Instruction cycle

In general, besides the parallelism in algorithms such as fingerprint enhancement algorithms, there can also be parallelism in instructions. The potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel. By exploiting these parallelisms, instructions can be pipelined resulting in improvement of performance. However, no matter using a hardware approach (hardware pipelining) or a software approach (software pipelining), the complexity of the architecture will be largely increased. And compared with the algorithm-level parallelism, in our case, instruction-level parallelism is more or less trivial. Due to these reasons, I give up exploiting the ILP by pipelining. And finally, in the simulation and synthesis results, the system bottleneck is proved to be in the memory control module instead of being lack of pipelining in the pixel processor.

Instead of pipelining in several stages, an instruction is executed sequentially in four stages as shown in the following figure.

Instruction fetch	Operand fetch	Process	Result store

Figure 4.14 Decomposition of instruction cycle

- Every new instruction is dispatched by the SIMD controller at the end of last instruction cycle. Thus it costs no time in instruction fetching. An instruction is executed with two operand addresses, one destination address indicating the address for the result both in the external memory or processor cache, a 16bit instruction code and an integer number called instruction number. To cope with the huge computational load of convolutions in fingerprint enhancement, a new parameter is introduced, that is, instruction number. Let's first look into the following situation: $A_1 \times B_1 + A_2 \times B_2 + A_3 \times B_3 + A_4 \times B_4 + \ldots + A_n \times B_n$ This accumulation requires n multiplications and n additions. However, addition takes much less time than multiplication. That's to say, in the process stage for executing an addition, the pixel processor are in idle most of the time. If we incorporate the multiplication and addition in a signal process stage, nearly half of the time cost will be saved despite that the instruction cycle may be prolonged a bit. Further, the store stage is avoided for the intermediate results in the accumulation. What a processor does in this case is to repeatedly do the multiplication and additions till all these products are accumulated. Let instructions number denote the number of recurrence, a large accumulations can be executed under a single instruction. For other operations don't require accumulations, an overhead of an addition with constant 0 at the end of the process is inevitable. And for a pure addition operation, the operands are directly passed to the adder. With instruction number, a single instruction can finish an accumulation of a sequence of recurrent operations or operands, saving a lot of

time and space for instruction code. It's obvious that in fingerprint enhancement the benefits overcome the overhead of slight prolonged instruction cycle time. And another advantage will be given later in the operand fetch.

- There three types of operands: immediate number, processor cache and processor memory. The first two types of operands can be fetched immediately because the immediate number is simply the operand address and the processor cache works as a register. As mentioned before, accessing pixel memory would cost an extra cycle. When the accumulation operands exist in the pixel memory, there will be a large time cost in memory accessing. By incorporating a large accumulation into a single instruction, the intermediate data store stages are avoided. Therefore, the data load stage can be simply overlapped with the process stage since there isn't a consideration on data dependence. And thus the extra fetch time is hidden.
- The process stage implements several arithmetic/logical operations needed in the enhancement algorithms. There are no control flow instructions like jumps and branches. That's because all the instructions are passively received from the SIMD controller instead of addressing and fetching. Of course, this simplification largely reduces the complexity of the hardware architecture. However, we lost the ability in managing the control flow of conditioning and branching. This problem will be solved in next section. An accumulation is bound at the end of process stage, and this can help incorporate accumulation operations into a single instruction despite slightly extending the instruction cycle.
- The last stage of result store writes the result into the processor cache or sets up a write single to the SIMD controller if the destination is in the external memory. When receiving the write single, the controller stores the results from all the processors in the output buffer and remembers the address. After all the instructions are carried out, the controller writes back all the data in the out buffer into the external memory.

The cycle per instruction (CPI) is dependent on the instruction type. For most of the instructions with immediate number and register operands, CPI equals 1.

4.2.4.2 Idle instruction

Till now, the pixel processor can deal with most of the instructions in fingerprint enhancement algorithm. However, two problems still exist before us:

i. **Control flow in branches.** As there are no control flow instructions, the instructions in all the branches are carried out.
ii. **Synchronization of the processors.** The instruction streams are dispatched by the SIMD controller to every processor synchronously. But if the processor finishes these instructions at different time, there will be a synchronization problem. This is possible since due to different conditions (data dependent), processors may jump into different branches.

As discussed in last section, the instruction streams are dispatched by the SIMD controller, so some of the control flow managements can surely be finished in the SIMD controller. This is possible only when all processors always jump into the same branches, which means they must have the same conditions and the control flow is identical to every processor. The variance of conditions in different processors leads to different branch selections. It not only forces us to finish the branch selecting in each processor but also brings us synchronization problem.

For the first problem, if all the braches are passed by, there will be no synchronization problem, because under any condition, the control flow will be the same. However, the instructions in undesired branches are also carried out leading to a wrong result. If the control flow remains the same but the instructions in unwanted braches are replaced by idle instructions, two problems are both solved!

Like in figure 4.15, when a > b, all the instructions in statement1 are replaced by idle instructions and when a < b, all the instructions in statement2 are replaced by idle instructions, the instructions stream will be fully in sequential and no jump and branch instructions are needed any more. In this way, each processor has to work longer than actually needed, but remember that, due to the synchronization problem, the processors with early finishes always have to wait for the latest one. Thus, there is no time overhead

but the complexity is much more simplified. The only job for the processor is to set an idle number into a special register based on the result of comparison instruction. The number is carried in the destination address of the comparison instruction. And the comparison result is forced to be written into another special register (somewhere in the processor cache). At the beginning of every instruction cycle, the idle number is check first and if it not equals to 0, the instruction is replaced by idle instruction and the idle number is decreased by 1.

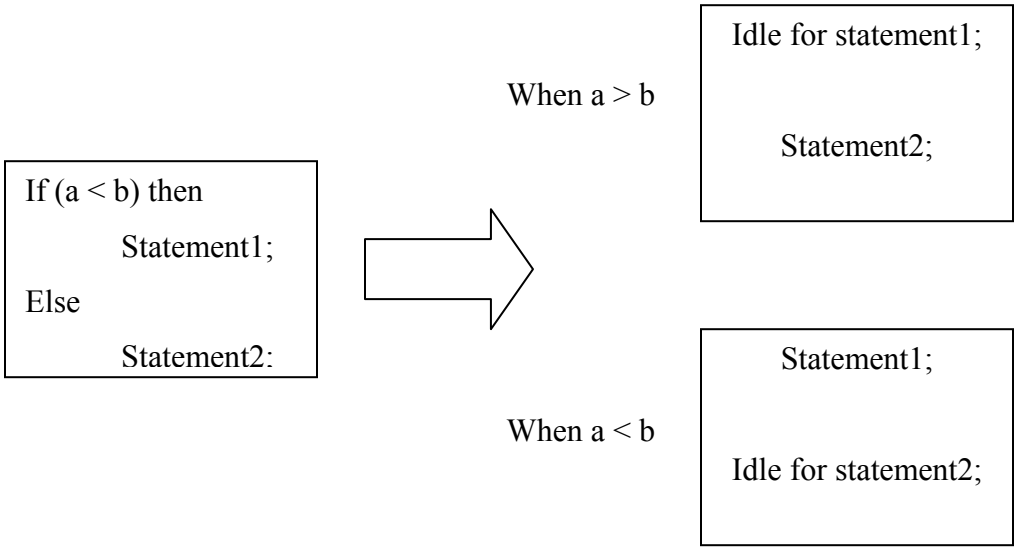


Figure 4.15 Idle instructions in branches

With this implementation, the codes need to be slightly changed and may be bit longer. Anyway, compared with the simplification in architecture, it's worthwhile.

All the original codes (C++ in my software simulation) should be complied into executable codes for this architecture. This is done by hand, because in general there is not a compiler for this special architecture.

4.3 Data representation

There is no Floating Point Unit (FPU) in the pixel processor. The abs cense of FPU not only reduced the hardware size but also leads to an improvement of performance, despite the limited lost in accuracy.

The use of fixed-point arithmetic in the embedded biometrics systems has been studied in [7], [8]. In our SIMD architecture implementation, we have adopted a similar approach to facilitate the whole process. A commonly used representation of a fixed-point arithmetic system is shown in Figure. 4.16. Here S denotes for unsigned or two's complement (signed). WL denotes for word length. IWL denotes for integral word length and FWL denotes for fractional word length.

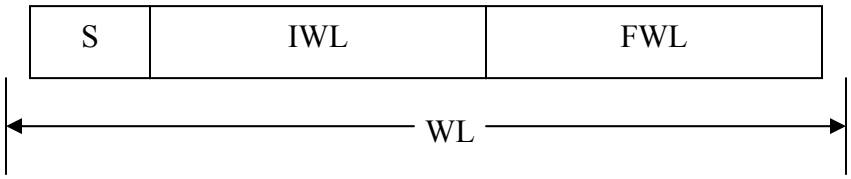


Figure 4.16 Representation of the fixed-point arithmetic system

The following table gives the accuracy analysis results for each step of the enhancement.

Enhancement step	**Fix-point arithmetic representation**	**Max error**
Normalization	8.8	1 gray level
Gaussian filter	15.16	1 gray level
Orientation estimation	3.13	0.0001
Frequency estimation	8.15	0.00003
Region mask	--	--
Gabor filter	8.8	1 gray level

Table 4.2 Accuracy analysis

As in table 4.2, by using a 15.16 representation, the computational error won't exceed the values listed above.

4.4 Conclusion

In this chapter, a parallel architecture called SIMD architecture is proposed. By using a processors array, the parallelism in fingerprint enhancement algorithm will be exploited sufficiently. And the data transmission between external memory and processor can be hidden by data pre-fetching. This architecture doesn't have FPU and instead, fix point representation is adopted.

Chapter 5 Implementation

After the architecture analysis and design, I implemented this pixel SIMD architecture in VHDL. Some of the arithmetic units are generated by Xilinx CoreGenerator.

The whole system consists of 19 VHDL files: PixelProcessor.vhd, PixelMemory.vhd, SIMD.vhd, clk_divider.vhd, div_32.vhd, sine.vhd, multiplier.vhd, slding_window.vhd, arctan.vhd, stdlib.vhd, sqrt.vhd, ALU.vhd, external_memory.vhd, PixelControl.vhd, memctrl.vhd, test_simd.vhd, adder.vhd, shared_alu.vhd, my_package.vhd. Some of them are only used for simulation and the rest can be synthesized into real circuit.

This chapter is organized in this way: section 5.1 presents the design for external memory; section 5.2 presents the design of memory control module; section 5.3 presents the design of clock divider; section 5.4 presents the design of pixel controller; section 5.5 presents the design of pixel memory; section 5.6 presents the design of pixel processor; section 5.7 presents the design of ALU; section 5.8 presents the design of SIMD controller; section 5.9 presents library used in our design; section 5.10 gives a global view of the whole implementation and section 5.11 draws to a short conclusion.

5.1 External memory

The external memory module is used to simulate the SRAM on the FPGA board from Avnet Design Services. There are up to 2MB onboard asynchronous SRAM and the memory bandwidth is 32bit. The highest operating speed is 80 MHz.

The memory size in this module is chosen to be 2^{19} = 524288 bytes, the minimum size for processing a 256 by 256 fingerprint image. As a module only for simulation, the external memory initially reads data from one or more binary files and at the end of the simulation, it can write back the results to a file on the disk. With this file, intermediate and final results can be examined.

External memory is implemented in external_memory.vhd and the following table gives the in and out ports.

Name	Description
addr[abits-1]	A abits-1 wide address both for read and write operation.
write	The write signal is used to indicate there is a write operation when it goes high and a read operation when it goes low.
idata[31:0]	32 bit wide input data signal.
odata[31:0]	32 bit wide output data signal

Table 5.1Ports in external memory

5.2 Memory control

Since a pixel processor doesn't have the ability to address in the external memory, reading and writing data in external memory must be managed in the SIMD controller. Usually, the size of input data for the processor array can be quite large and memory control module is a dedicated module for external data transmission control. Under the control of SIMD controller, this module reads the data in external memory by the 32bit wide data bus serially and a 16bit mask is generated for every 16bit or 32bit data piece. This mask indicates which processors in the processor array require this piece of data. By setting the corresponding bit to 1, the data will be dispatched to that pixel processor. In this case, redundant data reading is avoided since a single piece of data can be sent to all the processors with the corresponding mask bits set to 1. When all the mask bits are cleared to 0, the input data is ignored. In another word, the zero mask stands for the finish of external data reading.

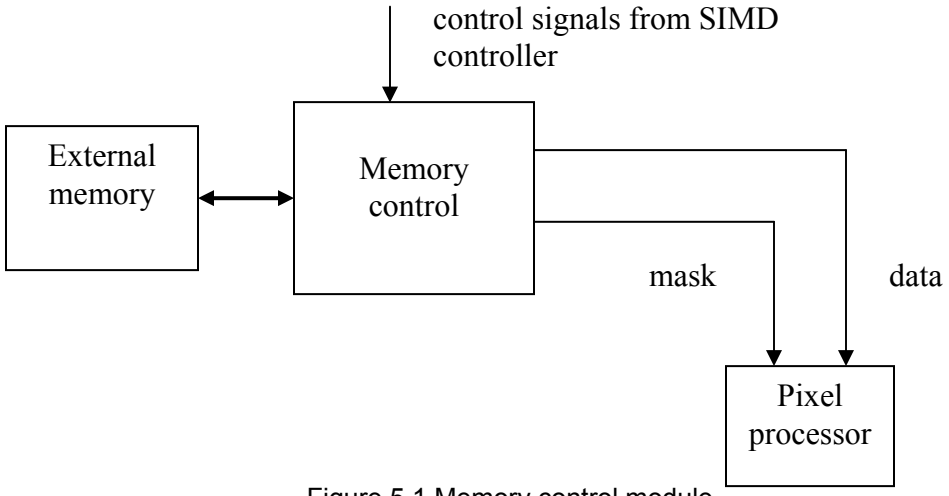


Figure 5.1 Memory control module

As analyzed in the chapter 2, in fingerprint enhancement, the required input data for each pixel always lies in a small data window centered at that pixel. The processor array is assigned to a group of pixels at a time, called pixel array. The small data windows of these pixels usually intersect with each other and form a bigger data window. All the data pieces (16bit or 32bit wide) in this data window are transferred from external memory to pixel processor serially by the 32bit wide data bus. Thus, the external data fetching is actually a burst transfer.

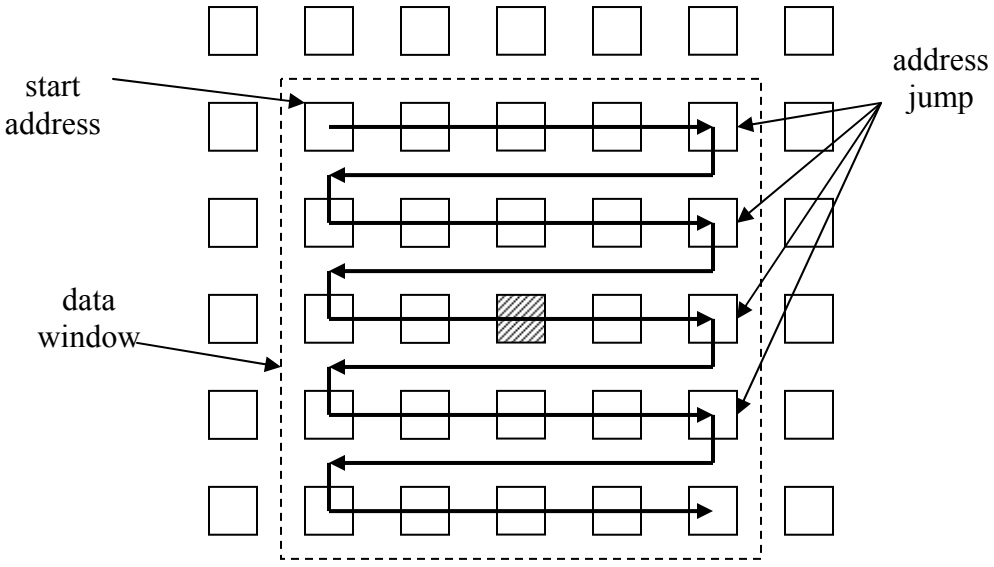


Figure 5.2 Burst transfer

In figure 5.2, let the blocks note for the grayscale values of the fingerprint image. Each block corresponds to a pixel grayscale value (data piece). For example, the Gaussian smooth filtering requires a 5 by 5 spatial convolution. The grayscale values in the window with dotted line (data window) are all required for the filtering operation for the pixel in the center, which is denoted in shadow. The 25 grayscale values are transferred in a burst mode in an order denoted by the arrows. Thus, only the start address and four address jump points are enough for the burst transfer operation. And it's the same for data transfer for a pixel array, with a bigger data window. Sometimes, several burst transfer operations are required in a single data load stage (Figure 4.11) and they are carried out sequentially.

Memory control module consists of two parts: memory controller and sliding window controller.

5.2.1 Sliding window controller

When the processor array is assigned to groups of pixels (pixel array) in the fingerprint image, it slides like a window from left to right and from top to bottom. In the same way, the data window enclosing the required input data also slide. The sliding window controller is designed to generate the correct data window for the corresponding pixel array. When the start address is given, the controller should generate every data window for each pixel array from top left to bottom right in the fingerprint image.

Two parameters decide the movement mode of the sliding data window: distX and distY, which describe the distance of two connected data windows (also pixel arrays) in x and y directions respectively. By setting the values of distX and distY, different data fetching modes can be selected according to the algorithms. As in figure 5.3, the dotted line encloses pixel array and the black box depicts the data window. For data window A and B, distX = 2, since there are 1 pixel array in the middle of the two and for A and C, distY = 1, which means they are closely near by. Later, in memory controller, the distance between two neighbor pixels is also needed. In fact, these two distances have a linear

relationship: the data window (pixel array) distance is P times of pixel distance, where P is pixel array length. Thus, these two kinds of parameters can be incorporated in order to save the ports.

Let D denote for the wide of data window and P denote for the wide of processor array. Note that when (D-P) > (distX-1) ×D or (D-P) > (distY-1) ×D, there will be an intersection between two data windows. That means redundant data reading happens! However, it is inevitable in this architecture for three reasons. Firstly, elaborated intersection check and management are needed to avoid such situation. The intersection can happen not only between two connected data windows but also two far away data windows due to the large window size. The complicated intersection check may greatly increase the hardware area and lower the performance of the whole system. Secondly, the overlapped data piece has different positions in the data windows, which means its address in the pixel memory must also be changed. For example, in figure 5.3, the shadowed data piece is the 33rd element in data window A, probably this piece of data will be stored in pixel memory with an address of 33. But in data window C, it's the first element and for processor array C, it should have an address of 1 in the pixel memory. Lack of reloading this piece of data into pixel memory requires tracking its position in the data window and changing its address in pixel memory correspondingly. This factor also brings a lot of overheads. Thirdly, larger pixel memory size may be needed to keep shared data pieces for later data windows. And the memory size varies according to different distX, distY and data window size. That is, it's algorithm dependent. As a matter of fact, as the size of processor array increases, the redundant decreases. When the number of processors equals the number of pixels in the image, there will be no redundant in data reading since there is only one data window!

The movement of the sliding data windows is controlled by two counters, couner_H and counter_V. When the data window moves one step from left to right, counter_V increases by 1. When counter_V reaches a threshold, which means the sliding window reaches the right side of the image, it is cleared to 0 and counter_H increase by 1. And when

counter_H reaches its threshold, the sliding window goes over all the image and all the pixels have been assigned to the processor array. The start address of data window is calculated by this formula:

Address = start address + counter_V × P × distX × W + counter_H × P × distY × S × W

Here, P is the width of the processor array, W is the data piece width (in bytes) and S is the width of the image.

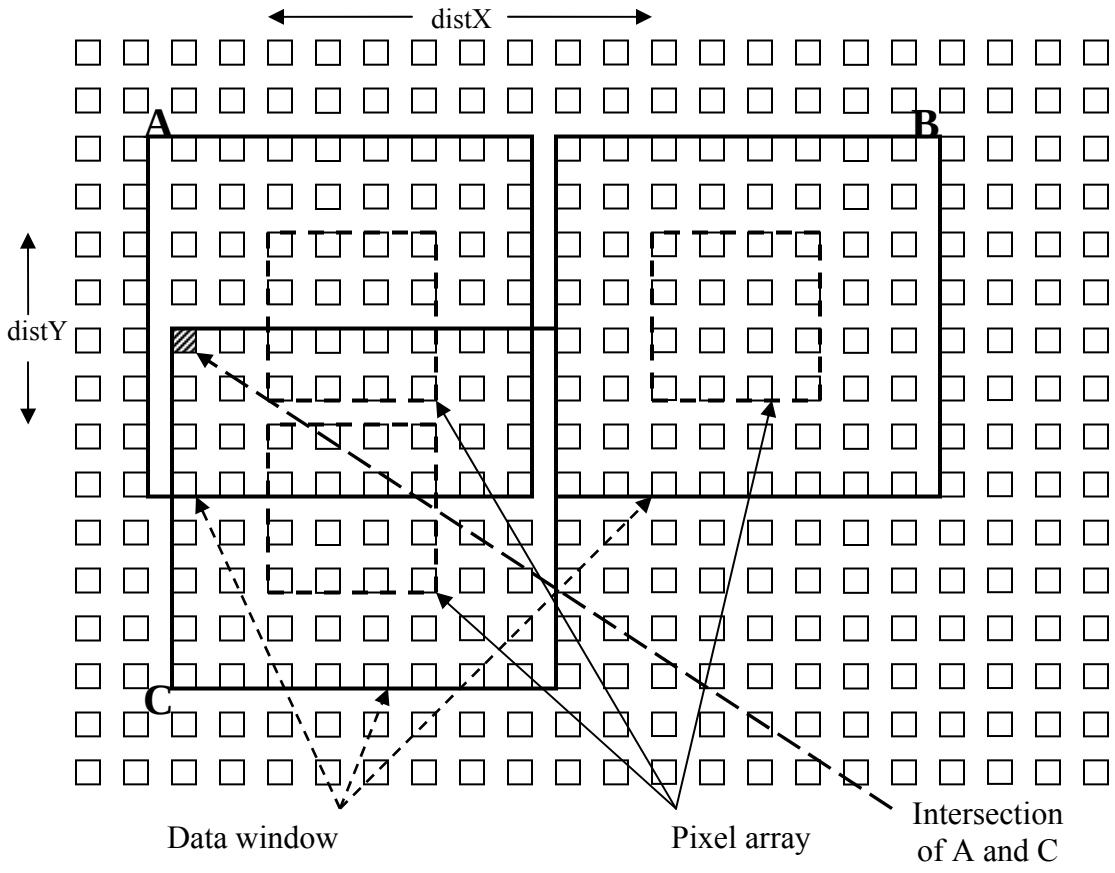


Figure 5.3 Sliding window

Another important task for sliding window controller is to do over boundary check. Unlike in software implementation where the memory overflows are handled by the compiler or run time system, the sliding window controller does the check in every step. There are four kinds of basic over boundary statuses in our case, as shown in following figure. Since each pixel requires the data pieces from its neighbors, it's possible that the data window of the pixel near the edge of the image may over the image boundary. Thus

the start address of the data window must be adjusted to point to valid data in external memory and the over boundary parts are usually forced to constants. This constant is algorithm dependent. For grayscale values, the constant is 255 (white pixels), assuming that these area in the image are white background and for orientation estimation, the gradients in the over boundary parts are all assumed to be 0, which also means they are white background in the image.

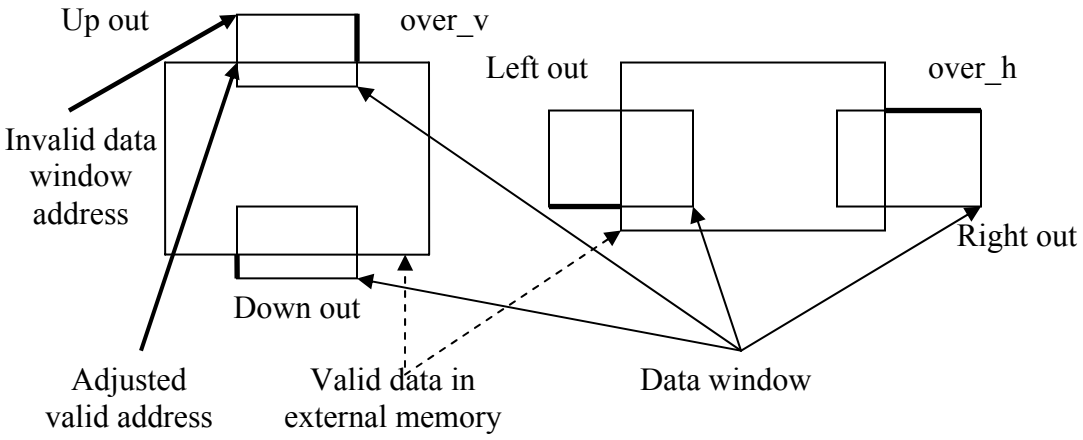


Figure 5.4 Data over boundary problem

As the valid address is sent to memory controller, another two parameters are also generated to indicate the size of the over boundary parts: over_ h and over_v. Over_v is the height of the over boundary part and over_h is the width. The signs of these two parameters indicates the over boundary directions, left or right and up or down respectively.

Sliding window controller is implemented in sliding_window.vhd and table 5.2 describes the in and out ports.

Name	Description
clk	Input clock signal.
rst	Input reset signal.
sin	Input sliding_in type signal.
Sliding_sel	Input enable signal for current data window.
Sliding_addr	Output address signal for next data window's enable signal.
ready	Output ready signal.
mrst	Output reset signal for pixel processor.

memi	Output mem_in type signal.

Table 5.2 Ports in sliding controller

Sliding_sel and sliding_addr signals are used to fetching the enable bit signal for each data window from a mask in SIMD controller, that is, sliding mask. Since in some algorithms, not all the pixel arrays in the image are involved, with the corresponding bit set to 0 in the sliding mask, the memory control module can jump over these pixel arrays. In fingerprint enhancement, the areas which are considered as background by region mask estimation can be hidden by the data window mask, leading to an improved result and higher performance.

Sliding_in is a record type of several control signals from SIMD controller.

- Block_sizeX: the width of the data window (pixel array data window).
- Block_sizeY: the height of the data window.
- Size: the product of image width (in pixel) and data piece width (in bytes).
- Cnt: number of steps of the sliding movement.
- Addr: initial address in external memory of the required input data.
- distX: distance of two neighbor data windows in x direction.
- distY: distance of two neighbor data windows in y direction.
- Conv_sizeX: pixel data window width.
- Conv_sizeY: pixel data window height.
- Read_cnt: number of burst operations.
- Memrdy: ready signal from memory controller.
- Membusy: busy signal from memory controller.
- Handshake: handshake signal from SIMD controller.

As mentioned that in a data read stage, more than one burst reading operations are probably required according to different algorithms. This means more than one data windows must by generated under different sliding_in type signals. Thus, except the read_cnt, memrdy, membusy and handshake signals in sliding_in type, the rest are all signal arrays with each element for a separate burst operation. And read_cnt is the array size. These burst operations are carried out sequentially in the data read stage. Membusy

signal informs the sliding controller that a burst operation has been finished and memrdy signal means all the operations are finished. Handshake signal is important to synchronize the data read stage and process stage. In this SIMD architecture, data read stage is overlapped with process stage by pre-fetching data for next process stage. But usually these two stages finish at different time and leads to synchronization problem. With handshake signal memory control module can wait for the pixel processor to finish the process stage.

Mem_in is a record type of input signals for memory controller. As sliding window controller generates the data window parameters, they are passed to memory controller.

- Write: write signal for write operations in external memory.
- Enable: enable signal for the burst operation.
- Block_sizeX: the width of the data window (pixel array data window).
- Block_sizeY: the height of the data window.
- Conv_sizeX: pixel data window width.
- Conv_sizeY: pixel data window height.
- Addr: start address of the burst operation.
- Size: the product of image width (in pixel) and data piece width (in bytes).
- distX: distance of two neighbor pixels in x direction.
- distY: distance of two neighbor pixels in y direction.
- Addr_w: starting address of the first burst operation in last data read stage.
- Over_v: over boundary parameter.
- Over_h: over boundary parameter.
- Read_cnt: number of operations in a data read stage.

5.2.2 Memory controller

Memory controller receives the start address from sliding window controller, reads all the data pieces in the data window from external memory and sends them to processor array. At the same time, the data mask is generated. Since the data bus is 32bit wide, if the data piece width is 16bit, 2 data pieces will be fetched at a time. Thus the mask is up to 32bit wide.

Similarly as the movement of sliding window, the burst read operation is under the control of two counters: counter_H and counter_V, each records the steps in x and y directions respectively. The difference is that, in burst operation, special attention is paid to avoid over boundary problem (memory overflow). With over_v and over_h signals generated in sliding window controller, the over boundary data piece is picked out and its address is set to 0, where in the external memory the desired value for over boundary parts is pre-stored there. Another difference lies in that because a data piece can be both 32bit and 16bit wide, the couner_h may increase by 1 or 2 in each step in x direction. When counter_h increases by 1, the address in external memory also increases by 2 byte (16 bits) correspondingly. To reduce the complexity of the hardware, every data piece is assumed to be 16bit wide, thus the incensement of counter_h is more regular. For every two data piece, the memory controller will generate the data address in external memory. A 32bit data fetch can be finished in the receive end (pixel controller) by combining two coterminous 16bit data pieces.

There is another problem due to 16bit data fetch. Memory controller read 32bit data from external memory each time, that is, two data pieces. If one of them is over boundary and the other is not, as shown in figure 5.5, we can not simply read the pre-stored value at address 0 because the other data piece will be lost! This situation is called half over boundary compared with those fully outside the boundary. In this case, the data pieces address is forced to be the beginning (A) or ending (B) of the corresponding rows. And finally, half word is cleared to the desired values for over boundary parts.

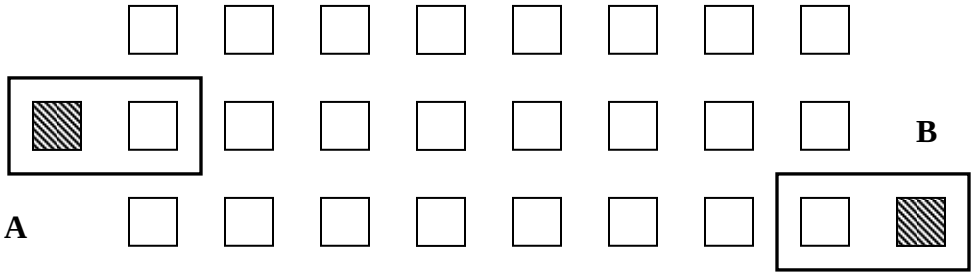


Figure 5.5 Half over boundary

When memory controller finishes one burst operation, a busy signal is sent to sliding controller. The sliding controller will generate next data window parameters for the next burst operation. When all the operations are finished, ready signal is set to 1 to inform sliding window controller that the current data read stage is over. The number of burst operations in a data read stage is carried in read_cnt signal.

Memory controller also generates data mask by checking if the current data piece falls in the private region of a pixel. The private region is decided by the index of this pixel in the pixel array and the distance between two neighbor pixels. Because we assume all the data pieces are 16bit wide, a 32bit data mask is generated in every data fetch. The correct data mask for a 32bit wide data piece fetch can be obtained by simple ignoring the higher 16 mask bits.

Memory controller is implemented in memctrl.vhd. Following is it in and out ports.

Name	description
clk	Input clock signal.
rst	Input reset signal.
ctrli	Input mem_in type signal.
idata	Input data for write operation in external memory
ctrlo	Output mem_out type signal.

Table 5.3 Ports in memory controller

Mem_out is a record type consists of output signal to SIMD controller.

- Odata: output 32bit signal.
- Mask: 32bit data mask.
- Ready: ready signal.

5.3 Clock divider

From [11], we know that the external SRAM can work at a speed at 80 MHz. This means our system frequency is limited to 80 MHz. However, the SRAM can not work at the same frequency with the whole system. This is because every time 32bit data is fetched from external memory, and thus the pixel controller needs two cycles to store them into

pixel memory in case the data piece width is only 16bit width. Thus, making external memory and the memory control module work at half frequency of the whole system, we can not only solve the 16bit data fetch problem but also boost the possible system frequency up to 160 MHz.

In this case a clock divider is needed to generate another clock signal whose cycle time is half of the system clock signal. This is quite simple by turning a counter repeatedly.

The clock divider is implemented in clk_divide.vhd. Table 5.4 describes the in and out ports in the clock divider.

Name	description
Clk_in	Input clock signal.
Rst	Input reset signal.
Clk_out	Output clock signal.

Table 5.4 Ports in clock divider

5.4 Pixel controller

Pixel controller is a bridge connecting pixel processor, pixel memory and SIMD controller. Every one pixel processor and two pixel memory modules are encapsulated in a pixel controller. And finally, 16 pixel controllers (pixel processor array) are connected with the SIMD controller.

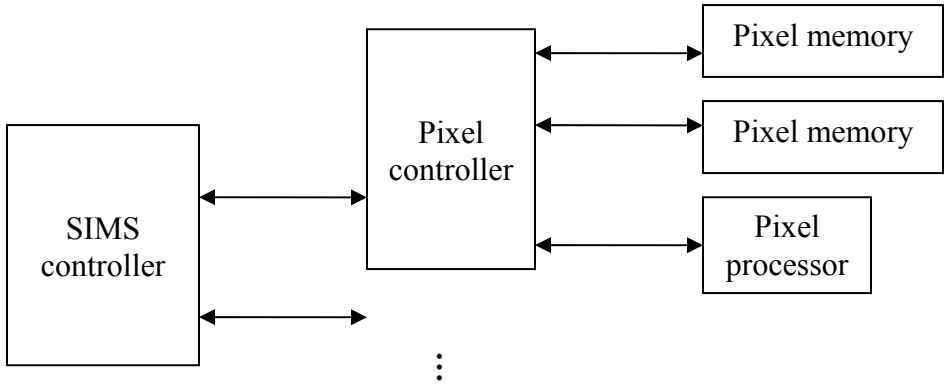


Figure 5.6 Pixel controller

In chapter 4.2.3, we have mentioned that in order to overlap the data read stage and process stage, data pre-fetching technique is introduced. In this technique, two pixel memories is required, one for the current process stage and the other for pre-fetching data for the next process stage. Thus, each pixel memories alternately play the roles of data destination for memory control module and data source for pixel processor. This is managed by pixel controller under the pmsel signal.

Another task for pixel controller is to receive data from memory control module and store into pixel memory. Not all the data pieces are picked up by pixel controller according to the corresponding data mask bit. Since external memory and memory control module work at a frequency half of the system frequency, pixel controller has two cycles to handle the data pieces in every fetch in external memory. If the data piece width is 16bit, there comes 2 data pieces each time. At the first cycle, pixel control checks the corresponding data mask bit in lower 16 bits and store the lower 16 bits data into pixel memory while in the second cycle, it checks mask bit in higher 16 bits and store the higher 16 bits data into pixel memory. If data piece width is 32bit, pixel control just checks the corresponding mask bit in the lower 16bits and store 32 bits data into pixel memory at the first cycle and idles in the second cycle. The data piece width is decided by transfer_size signal. Because not both of the two data pieces (when data piece width is 16bit) are required by the pixel processor in some cases, we can not combine the two 16bit wide data pieces and store it in the same way as when data piece width is 32bit. This may slightly increase the complexity of the pixel controller and pixel memory.

After every store operation, the data address in pixel memory increases by 1 and it's cleared to 0 at every beginning of the data read stages, witch is decided by wrst signal. This means data in pixel memory can not be hold to the next process stage since they are covered by the new data. Due to this reason, redundant data reading may happen sometime, and it accords with the analysis of redundant data reading between data windows in chapter 4.2.3. However, we at least benefits from two advantages in this architecture. First, we don't need larger pixel memory size to keep data for later process.

And what's more, keeping data in pixel memory requires more complicated address management.

Similarly with data store into pixel memory, a pixel processor can read both a 16bit wide data piece and a 32bit one. This is decided by wsel signal. Note that, till now this signal controls all the burst read operations in a data read stage, which means all these operations must have same data width. This is because in fingerprint enhancement, it dose not require read operations with different data width. But it's easy to do slight changes in the architecture to support read operations with different data width.

Now there are two signals for controlling the data width in data store and data read operations, transfer_size and wsel. This means fetching data from external memory into pixel memory and read data from pixel memory by pixel processor can be in different modes. In this case, there is more flexibility when coping with different algorithms.

Pixel controller is implemented in pixelcontrol.vhd. Table 5.5 gives the in and out ports.

Name	Description
clk	Input global clock signal.
clk0	Input clock signal with half frequency of the clk signal.
rst	Input reset signal.
wrst	Input reset signal for data read stage and process stage.
pmsel	Input signal for selecting pixel memory.
transfer_size	Input signal indicating the data piece width from memory control module.
instr	Input 16bit wide signal carrying the instruction code for pixel processor.
mask	Input 32bit wide data mask.
idata	Input 32bit wide data signal from memory control module.
addr_a	Input 32bit wide address signal for operand A.
addr_b	Input 32bit wide address signal for operand B.
addr_r	Input 32bit wide address signal for destination.
alu_in	Input 48bit wide data signal from external ALU.
neighbor	Input 32bit wide data signal from neighbor pixel processor.
num	Input integer signal carrying the number of operations.
wsel	Input 2bit wide signal indicating the data piece width read by pixel

	processor.
p	Input integer signal indicating pixel processor's index number in processor array.
odata	Output 32bit wide data signal for process result.
w	Output write signal indicating a write operation to external memory.
w0	Output write signal indicating a write operation to sliding mask.
rdy	Output ready wide signal for process stage.
alu_a	Output 32bit wide signal carrying operand A for external ALU.
alu_b	Output 32bit wide signal carrying operand B for external ALU.
ctrl	Output 2bit wide signal for processor communication control.

Table 5.5 Ports in pixel controller

5.5 Pixel memory

Pixel memory is a piece of exclusive memory for a pixel processor. All the data pre-fetched from external memory are stored in pixel memory for the next process stage. This can largely release the limitation due to the single data bus connected to external memory.

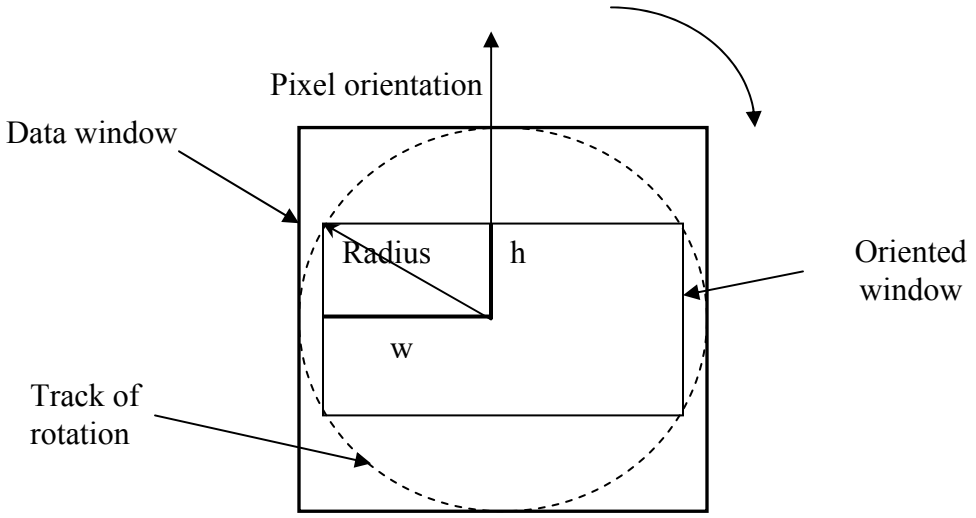


Figure 5.7 Data window in frequency estimation

The size of pixel memory is algorithm dependent. In fingerprint enhancement, frequency estimation requires the most pixel memory size. As mentioned in chapter 2.2.2.5, the oriented window for frequency estimation consists of 32 × 16 data pieces (16bit wide grayscale values). However, this oriented window is vertical to the orientation of the

central pixel. This implies that the window may be rotated by any degree according to the actual orientation at that central pixel. Thus in order to preload all the possible grayscale values for the process stage, all those in the dotted cycle must be included in the data window (a data window is always a square in our SIMD architecture). The data window width can be calculated by the following formula:

In algorithm implementation, the data window is usually located around the central pixel symmetrically. Thus, the width of the data window must be an even integer, including the central pixel and two equal parts in left and right side. And in the data window there are $(34+1)^2 = 1225$ grayscale values. Besides grayscale values, another burst read operation is also required in data read stage to fetch the orientation value of the central pixel. This will occupy only 16 bits in the pixel memory. Totally the required pixel memory size is 1225 + 1 = 1226 half words.

For most of these algorithms, 16 bits are enough to hold the data piece and in some others like mean value estimation, the read and write operation in external memory involves 32bit data pieces, but actually only a little memory space is enough for these data in fingerprint enhancement. In this implementation I choose to allocate 16 words memory space for 32bit operations in pixel memory. Thus the size of the pixel memory is $1226 \times 2 + 16 \times 4 = 2516$ bytes and for a 4 by 4 processor array, the total memory size is $2516 \times 2 \times 16 = 80512$ bytes.

There are up to 720k bits (90k bytes) block RAM and 320k bits (40k bytes) distributed RAM on Spartan 3 XC3S2000 chip. Thus the block RAM can be used to build up pixel memory and the distributed RAM can be used for caches/registers and other arithmetic units. Each pixel memory consists of 1226 16bit wide blocks and 16 32bit wide blocks. By chip selecting signals sel0 and sel1, 16bit or 32bit blocks can be chosen for the operations.

Compared with input data, the results of process stage are much less for each pixel. However, there are only 16 processors and for an image with 256 by 256 pixels, each processor have to work for 256×256/16= 4096 pixels sequentially in every algorithm. It's obviously that there is not enough space in pixel memory to hold these 4096 results until the finish of algorithm. All these results must be stored into external memory immediately after the process stage (in the process stage the data bus is probably being used by data pre-fetching). Thus they are written to the output buffer in SIMD controller instead of the pixel memory.

As to those intermediate results in the process stage, the cache in the pixel processor is more suitable than pixel memory to store. That's because the on chip block RAM is organized as asynchronous blocks, data fetching and storing in pixel memory will cost an extra cycle. Therefore, neither the final results nor the intermediate results of the process stage will be stored in pixel memory. The pixel processor only read data from pixel memory.

It's possible that the pixel processor may fetch two data pieces in the pixel memory at one time if both of the two operands are in it. To avoid one cycle delay in this case, pixel memory must be dual-port.

Pixel memory is implemented in PixelMemory.vhd and the following table gives the in and out ports.

Name	Description
clk	Input clock signal.
WE	Input write enable signal.
sel0	Input chip selecting signal for port0. 16bit and 32bit operations can be selected by this signal.
sel1	Input chip selecting signal for port1. 16bit and 32bit operations can be selected by this signal.
data_in	Input 32bit data signal from pixel controller.
addr0	Input integer address signal for port0.
addr1	Input integer address signal for port1.
data0	Output 32bit data signal for port0.

data1	Output 32bit data signal for port1.

Table 5.6 Ports in pixel memory

5.6 Pixel processor

5.6.1 Instruction opcode

Pixel processor executes all the arithmetic/logic operations according to the opcode sent by SIMD controller. The processor dose not have the ability to access external memory and all the operands are stored in pixel memory or caches in pixel processor. The operand addresses are carried in signal addr_a and addr_b. Sometimes the operands are immediate numbers directly sent by SIMD controller and still they are carried in these two signals. The destination address for the result is carried in add_r. It can be an address in external memory, pixel memory or processor cache. All the types of operands and result destinations are decided by the auxiliary bits in the opcode.

An opcode is 16bit wide and it format is given as follows:

opcode:

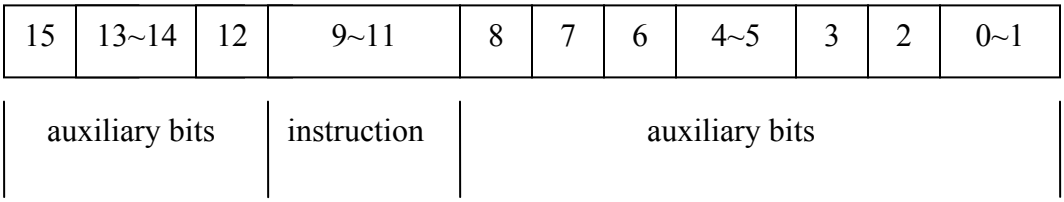


Figure 5.8 Opcode format

- Bits 0~1: these two auxiliary bits indicate the type of operand A, 00 for pixel memory, 01 for processor cache and 10 for immediate number.
- Bit 2: this bit indicates the length of operand A, 0 for 16bit and 1 for 32bit. This bit is only valid for data fetch in pixel memory.
- Bit 3: this bit indicates the type of operand address A. Like in C/C++, the address reference can be both by value and by address. When this bit is set to

0 operand address equals the value in addr_a and when it is 1, the address actually equals the value in pixel cache at address of addr_a.

- Bits 4~5: these two auxiliary bits indicate the type of operand B, 00 for pixel memory, 01 for processor cache and 10 for immediate number.
- Bit 6: this bit indicates the length of operand B, 0 for 16bit and 1 for 32bit. This bit is only valid for data fetch in pixel memory.
- Bit 7: this bit indicates the type of operand address B. When this bit is set to 0 operand address equals the value in addr_b and when it is 1, the address actually equals the value in pixel cache at address of addr_b.
- Bit 8: this bit indicates the type of result destination. When it is set to 0, the result will be stored in processor cache and when it is 1, the result will be stored in external memory.
- Bits 9~11: these three bits indicate 8 types of different arithmetic/logic operation, 000 for 32bit division, 001 for 32bit addition/subtraction, 010 for 16bit square root operation, 011 for 16bit multiplication, 100 for 16bit sine/cosine operation, 101 for 32bit comparison, 110 for 32bit shift operation and 111 for 32bit arctangent operation. When all the 16 bits of the opcode is set to 1, it means an idle instruction and the pixel processor will be idle for a cycle.
- Bit 12: this bit indicates the sign of accumulation, 0 for adding and 1 for subtracting.
- Bits 13~14: these two bits indicate 4 modes of writing the results stored in output buffer into external memory (before results are stored into external memory, they are stored in the output buffer in SIMD controller until the process/read stage is finish). When the two bits are 00, 16bit wide results from the 16 processors are written to external memory sequentially. When 01, first join two 16bit wide results from two near instructions that both require writing results into external memory, and then write the 16 32bit wide results into external memory. This mode can allocate different results together in external memory, which brings convenience for some algorithms. When 10,

32bit wide results from the 16 processors are written to external memory sequentially. And when 11, only the 32bit result from processor 0 will be stored into external memory.

- Bit 15: this bit indicates the communications among pixel processors. When it is 0, there is no request on communicating in processors and when it is 1, dynamic connections will be set up among these processors as described in chapter 4.2.2. These connections will be set up in the follow four cycles in a predefined order. In fingerprint enhancement, this function is used to accumulate the results from the 16 pixel processors into processor 0.

5.6.2 Instruction execution timing

Every instruction is executed in three stages in the pixel processor (see figure 5.9). These three stages are different from those three mentioned in chapter 4.2.3. The latter stages are in algorithm level for the processor array while these stages are in instruction level for a single pixel processor.

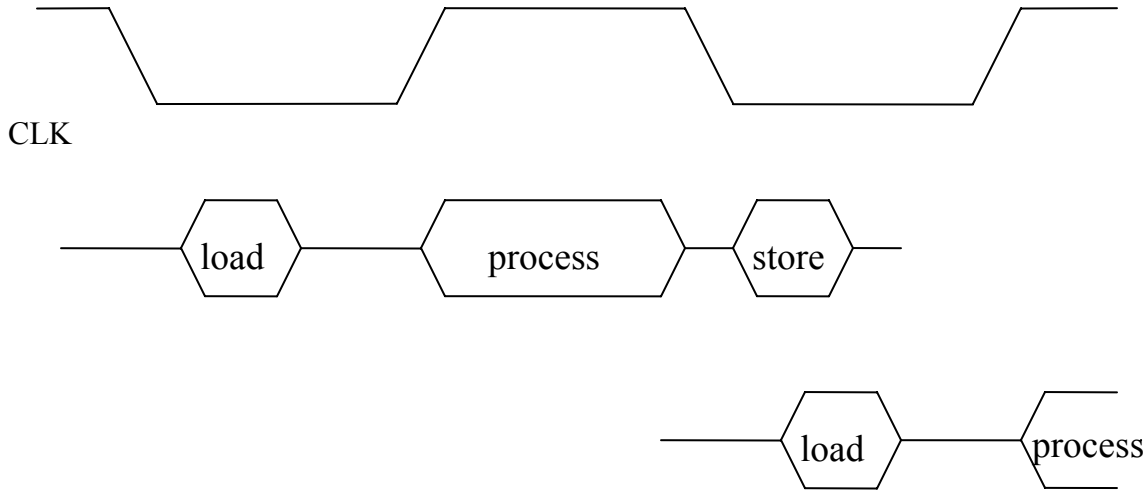


Figure 5.9 Instruction execution timing

The load stage reads operands from pixel memory or processor cache, or simply gets the value in operand address signal when the operand is an immediate number. This stage begins right after the falling edge of the clock signal, since new opcode is sent at that time by SIMD controller. Process stage begins at the rising edge of the clock signal and store stage at the next falling edge of the clock signal. It seems that the store stage is overlapped with the load stage and this may probably causes a WAR (write after read) hazard. This hazard may lead to wrong operand fetch in case there is data dependence between two instructions. Since the intermediate results are all stored in pixel cache, the WAR hazard can only happens when loading and storing in the same piece of cache. However, the load stage is not driven by the clock signal directly. And instead, at the falling edge of clock signal, SIMD controller first generates new opcode and addresses and then when these signals reach pixel processor, the load stage starts. Because the pixel cache is built up by distribute RAM, the access time is so trivial that there is enough time gap between the store stage and next load stage.

And as described in chapter 4.2.4.1, when instruction number is larger than 1, the intermediate store stages are avoided and the load and process stage can be overlapped. This brings big benefit in time cost and opcode size for large accumulation operations. But since it is a single instruction, there is only one pair of operand addresses. All the input operands for all accumulation operations are limited to be stored conterminously in pixel memory or processor cache. The address sent by SIMD controller indicates the start address and the processor will automatically increase the address by 1 at each operation until all the operands are fetched. This task is finished in address generator, which also copes with data fetching by indirect address when bit 3 or bit 7 in opcode is set to 1.

In chapter 4.2.4.2, we've given a solution for remedying the disability in control flow management. By inserting idle instructions in undesired braches, the control flow and synchronization problems are both solved. There are two kinds of idle instructions for pixel processor, static idle and dynamic idle. Static idle instruction is sent by SIMD controller with all 16 bits in opcode set to 1. Dynamic idle requires slightly more

complicated control in the processor since it is unknown at the compile time. At the beginning of every process stage, the processor checks a special counter, the idle counter. If this counter is larger than 0, the processor idles for 1 cycle and decreases the counter by 1. This counter is set by output controller according to the result of comparison operation which checks the condition of a branch flow. The number of idle cycles is carried in addr_r signal, since the length of the statement in branches is known at compile time.

At the store stage, the output controller writes the result into processor cache if bit 8 in opcode is 1, otherwise, w signal is set to high to inform SIMD controller the output must be stored in output buffer and then to external memory. The output controller also renews the idle counter and sets another signal, w0, to SIMD controller. The w0 signal indicates that the result should be written into sliding mask.

For most of the operations with instruction number of 1, CPI equals 1. But there are two exceptions:

- If at least one of the operands is in pixel memory, another clock cycle is needed in load stage. This can be solved by increase the instruction number by 1 and disable the data process/store stages in the first cycle.
- For operations like division, square root and sine/cosine, it takes more than one cycle to finish the process stage. Thus the process stage must be prolonged. In this case a special counter is used to track the end of these operations.

5.6.3 Processor structure

As shown in figure 5.10, a pixel processor consists of 6 parts: instruction decoder, address generator, processor cache, ALU, adder, output controller. The decoder reads the instruction from SIMD controller and dispatches control signals to the rest 5 parts. Address generator generates correct address for data fetch in pixel memory and processor cache. The processor cache is built up with on chip distributed RAM, with a size of 300 $\times$32 bits. Thus, $16\times300\times4$=1.92k bytes processor cache is needed for the processor array. This size is decided by Gabor filtering algorithm which involves large intermediate results and pre-computed values. ALU and the adder can carry out arithmetic/logic operations. In figure 3.10, we can see, besides ALU in the processor, an external ALU is also used. This is because for the three arithmetic operations of division, arctangent and sine/cosine, the arithmetic units require large hardware area. It is impossible for each processor to have exclusive units. In this case, a shared ALU works for every processor sequentially. Large gains in hardware area is achieved at a cost of prolong the process stage by 16 times for these three operations. The output controller controls the result output according to the opcode.

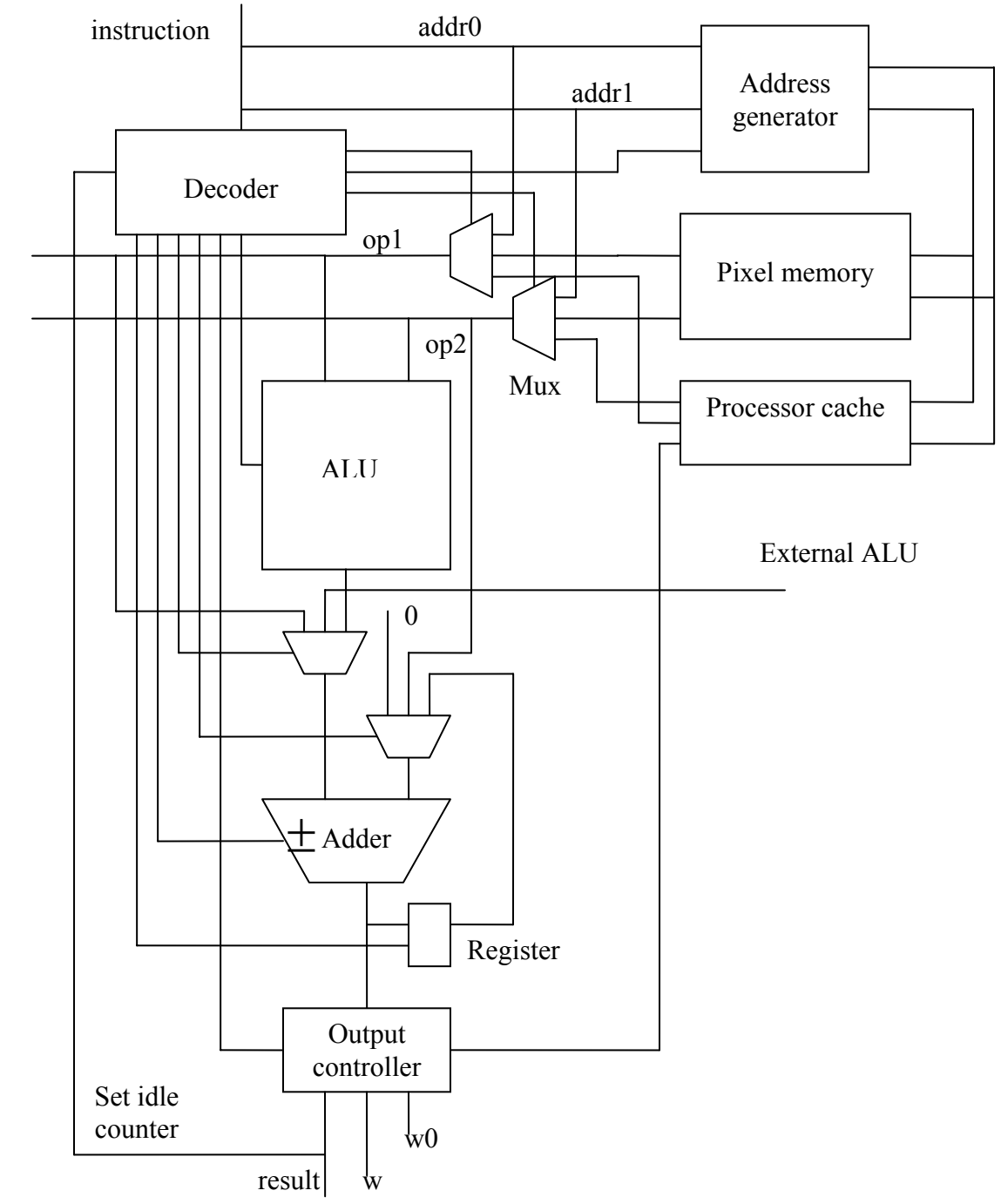


Figure 5.10 Pixel processor structure

Table 5.7 gives the in and ports of pixel processor and it's implemented in PixelProcessor.vhd.

Name	description
clk	Input clock signal.
instruction	Input 16bit opcode signal.
PM0	Input 32bit data signal from pixel memory.
PM1	Input 32bit data signal from pixel memory.
addr_a	Input 32bit address signal for operand 0.
addr_b	Input 32bit address signal for operand 1.
addr_r	Input 32bit address signal for output result.
alu_in	Input 48bit data signal from external ALU.
neighbor	Input 32bit data signal from neighbor pixels.
num	Input integer signal for instruction number.
rst	Input reset signal.
p	Input integer signal for processor index.
addr_PM0	Output integer address signal for pixel memory.
addr_PM1	Output integer address signal for pixel memory.
sel0	Output chip selections signal for pixel memory.
sel1	Output chip selections signal for pixel memory.
alu_a	Output 32bit data signal for external ALU.
alu_b	Output 32bit data signal for external ALU.
rdy	Output ready signal.
ctrl	Output 2bit control signal for dynamic processor communications.
w	Output write signal for output buffer.
w0	Output write signal for sliding mask.
data	Output 32bit data signal for result output.

Table 5.7 Ports in pixel processor

5.7 ALU

There are two kinds of ALU, on processor ALU and external ALU. The latter also can be called shared ALU, because it's share by the 16 processors in the processor array. These two units have similar architecture except the input and output architecture.

5.7.1 On processor ALU

Four operations can be executed by the exclusive ALU in every pixel processor. They are square root, multiplication, comparison and shift operations.

- Square root: this is a single operand operation. It is required in variance estimation and sometimes it can be used to scale operands in case that they are wider than the input ports of the arithmetic units. The on processor ALU (ALU for short later) has a 16bit wide square rooter with a latency of 0 which means an square root instruction can be executed within one cycle.
- Multiplication: the ALU use the on chip dedicated multiplier thus it's fast enough to finish a multiplication instruction in a single cycle. The multiplier support 16bit wide input and 32bit wide output.
- Comparison: with a 32bit wide comparator, the ALU returns 1 if operand 0 is larger than or equals to operand 1, other wise returns 0. It's similar with an add/subtract operation and can be executed within one cycle.
- Shift: The operand to be shifted is carried in addr_a signal and the number of bits to shift is carried in addr_b signal. However, it's impossible to use all the 32 bits in addr_b for a shift number. The 31 and 30 bits are used for control. If bit 31 equals 1, it means a right shift operation, otherwise it's a left shift operation. When bit 30 is set to 1, the shift in bits equal to the highest bit of shift operand and when it's 0, the shift in bits are forced to be 0.

The structure of ALU is described in the following figure.

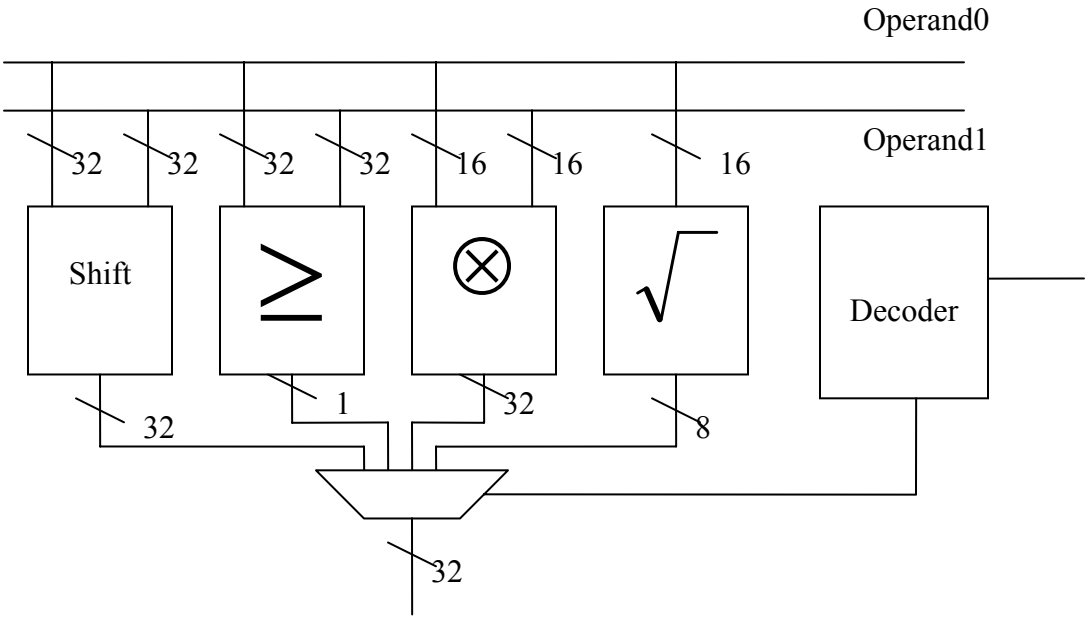


Figure 5.11 Structure of ALU

The ALU is implemented in ALU.vhd and table 5.8 describes it in and out ports.

Name	description
a	Input 32bit data signal.
b	Input 32bit data signal.
instr	Input 3bit instruction code.
o	Output 32 bit data signal.

Table 5.8 Ports in ALU

5.7.2 Shared ALU

In shared ALU, three arithmetic operations are executed here, arctangent, division and sine/cosine.

- Arctangent: shared ALU has a dedicated unit for arctangent operation. The input x and y coordinates are 32bit wide and the output is a16bit wide angle in radians. The two 32bit inputs are expressed as signed 1QN number (with 1 bit for the sign, 1 for integer part and 14 for the fraction part) and the output is expressed as signed 2QN number. It takes 18 cycles to execute the operation for 16 processors.

- Division: a 32bit divider is used to generate the quotient and remainder. The quotient is stored in the low 32 bits of the output while the remainder is stored in the higher 16 bits. It takes 68 cycles to finish for 16 processors.
- Sine/cosine: these two results are generated together by a dedicated unit. The input angle (in radians) is 16bit wide expressed as 2QN numbers and the output results are two 16bit wide numbers expressed as 1QN format. It takes 18 cycles for 16 processors.

Thus the output of arctangent operation can be directly sent to the sine/cosine unit without any format conversion. While the sine and cosine values are stored together in the 32bit result, shift operation can simply separates them.

The shared ALU has a similar architecture with on chip ALU and the only difference is that in the process stage, it works for 16 pixel processors instead of a single one. Therefore, there are 16 groups of operands arriving at the shared ALU. In pixel processor, when an instruction requires using shared ALU, the operands are sent at an open cycle in alu_a and alu_b signals while in other cycles these two signals are cleared to 0. An open cycle means the shared ALU is open for that pixel processor and waits for its input. Every processor has a different open cycle according to its index in the processor array.

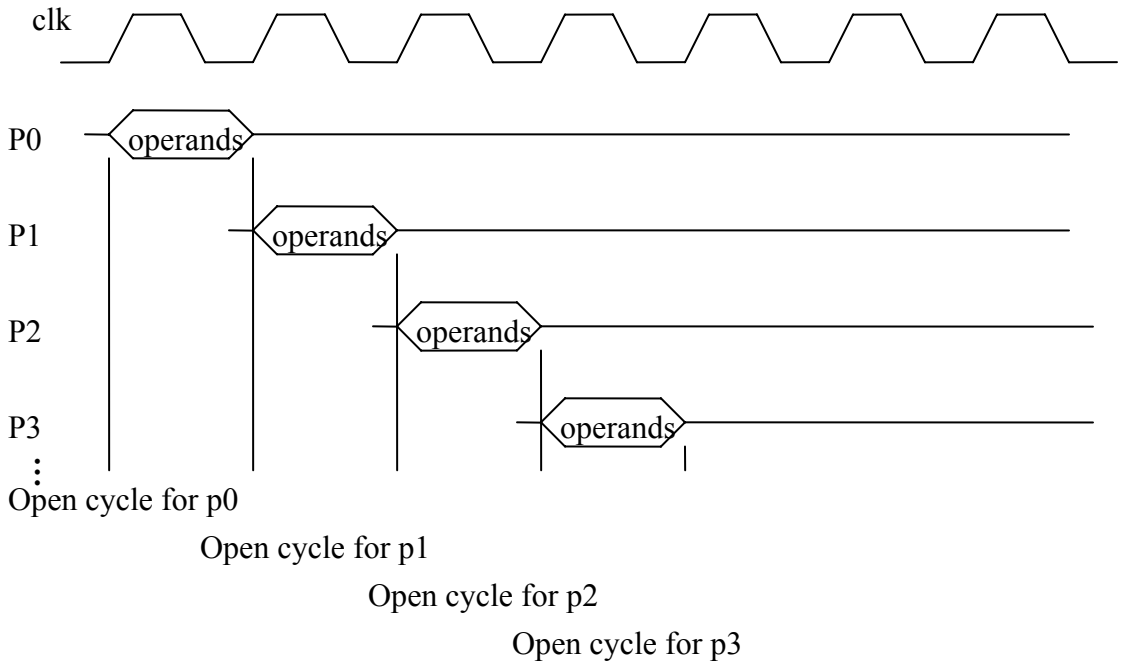


Figure 5.12Operands transfer timing for shared ALU

At the receive end, the operand controller combines these signal into a single one by simply logic and all the same bit in different signals. Similarly, the output from shared ALU changes periodically, with each cycle for a pixel processor. All the processor needs to do is to pick up the result at a corresponding cycle.

Due to synchronous problem, the processor which gets the result from shared ALU earlier must wait for the last one. In fact the solution of a shared ALU is trade for smaller hardware area with acceptable extending in time cost.

As shown in figure 5.13, shared ALU has one more module than on processor ALU, that is, address controller.

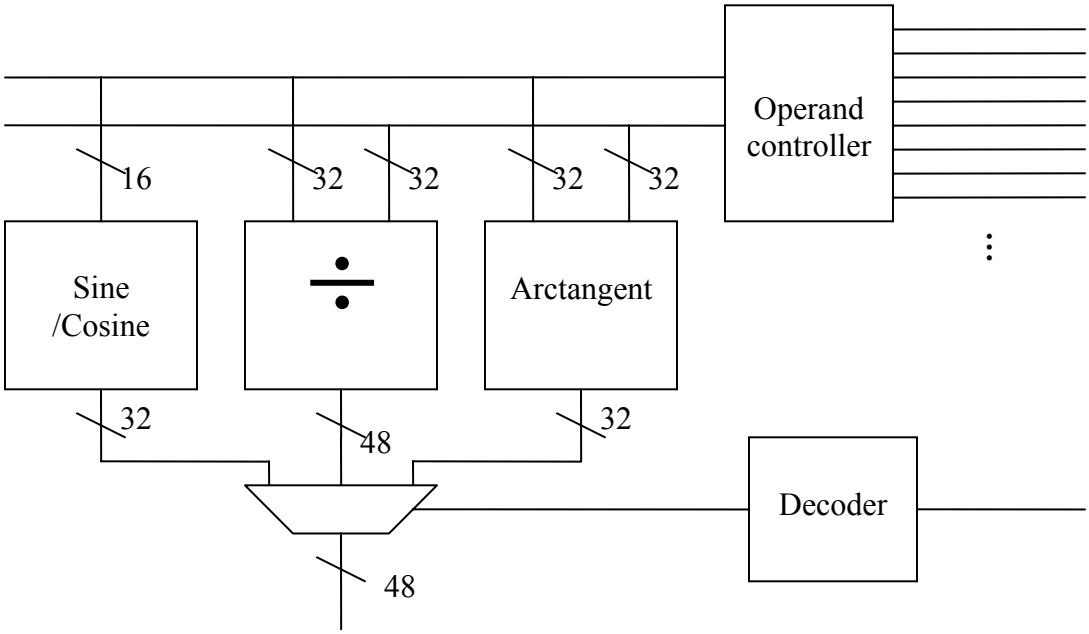


Figure 5.13 Structure of shared ALU

The shared ALU is implemented in shared_alu.vhd, the following table shows it in and out ports.

Name	Description
clk	Input clock signal.
a	Input vector array signal.
b	Input vector array signal.
instr	Input 3bit instruction opcode.
result	Output 48bit result signal.

Table 5.9 Ports in shared ALU

- A vector array is an array of 16 signals with a length of 32bit.

5.7.3 Arithmetic and logic units in ALU

Except the shifter, all the arithmetic and logic units in ALU for executing different operations are generated by XilinxCore generator. They include:

- 32bit adder
- 32bit divider

- 16bit square rooter
- 16bit sine/cosine
- 32bit arctangent
- 16bit multiplier
- 32bit comparator

Most of them make the arithmetic/logic operations in our SIMD architecture the same as in a software implementation. The only exception is the 32bit arctangent unit, which may lower the accuracy due to quantization error inside it. In the Cordic data sheet [13] from Xilinx, it says that *the accuracy of the output angle from CORDIC vector translation algorithm is limited by the number of significant magnitude bits of the input vector (X, Y).* That means the larger the input operand is the more accurate the output angle is. In fact, the actual input for the arctangent operation is Y/X, where x and y are the input coordinates. This means when X and Y increase the same times, the actual input value Y/X keeps the same but according to the CORDIC vector translation algorithm, the accuracy of the output is improved. Two examples are given in the following table.

Input number	**Result from ALU**	**Actual result**	**error**
1	0.9577	1.1071	0.1494
2			
10^8	1.1070	1.1071	0.0001
2×10^8			
2×10^4	0.4594	0.4636	0.0042
10^4			
2×10^8	0.4636	0.4636	0
10^8			

Table 5.10 Accuracy analysis in arctangent

Therefore, a special check and adjustment in input X and Y coordinates can greatly reduce the quantization error. This step is done at compile time so that the hardware architecture can keep unchanged, at a cost of adding a couple of instructions before the

arctangent instruction. The following are adjusted instructions for the arctangent operation:

```
If X<65535
    If Y<65535
        X = X×65535 (left shift by 16 bits)
        Y = Y×65535
    End if
End if
Arctangent(X, Y)
```

5.8 SIMD controller

SIMD controller is the main module that connects all others modules and organizes them working together, as shown in the following figure.

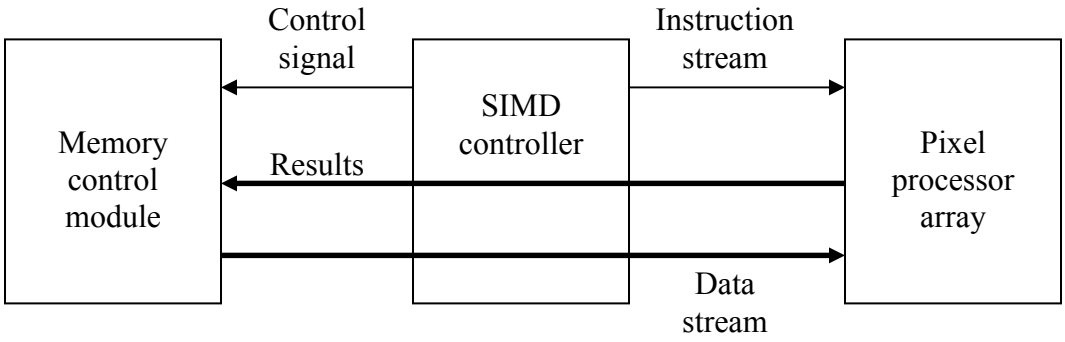


Figure 5.14 SIMD controller

5.8.1 Instruction and data streams

Two main tasks for SIMD controller is to control the generation of data stream and dispatch instruction stream to processor array, each under the control of a state machine respectively.

Let's first go over the data and instruction stream flow in this SMID architecture. Pixel processor array is assigned to a pixel array in the fingerprint image repeatedly until all the

pixels are processed. At the same time, the data streams (data pieces in the data window) for every pixel array are generated and sent to the processors. In each assignment, an algorithm is finished for a pixel array by sending a sequence of instructions to the processors. The reason why only one algorithm is finished in an assignment is that different algorithms require different data streams and they need to be generated again. The memory control module can generate data streams for all the pixel arrays by sliding the data window once the parameters of the first data window (for the first pixel array) are sent by the SIMD controller. This means that the SIMD controller needn't generate the data window parameters for every pixel array. However, it has to send the instruction steams repeatedly to the processor array although in every assignment, they are completely the same.

Therefore, two timing signals are needed to control the data stream and instruction stream, wrst and srdy. The former implies an algorithm is finished for a pixel array and the latter implies the algorithm is finished for all the pixels in the image and a new one starts. When wrst is high, the process stage and data load stage start and when it is low, the data store starts. When there comes a rising edge of the srdy signal, the SIMD controller generates new data window parameters and instruction sequence for the new algorithm and at the same time it also increase the algorithm_counter by 1. This counter is used to indicate different states in the state machines that generate data stream control signal and instruction streams. As in this SIMD architecture data pre-fetching technique is introduced, data process stage and the next data load stage are overlapped. However, in the first data load stage, there is no process stage. This means for a pixel array the process stage starts one stage later than the load stage. This is controlled by a start signal to make the processors start working after the arrival of data stream.

All the algorithms in fingerprint enhancement are complied into binary codes in a special format for this SIMD architecture and stored in the SIMD controller. These binary codes include 16bit opcode, 32bit addresses for operands and result and an integer for

instruction number. The generation of instruction streams is simply picking up different pieces of these codes and sending to the processors.

Figure 5.16 shows the data flow of data stream and instruction stream generation as described above.

In the process stage for an algorithm, each instruction dispatch is also under control of a signal, prdy, indicating the processor has finished last instruction (see figure 5.15).

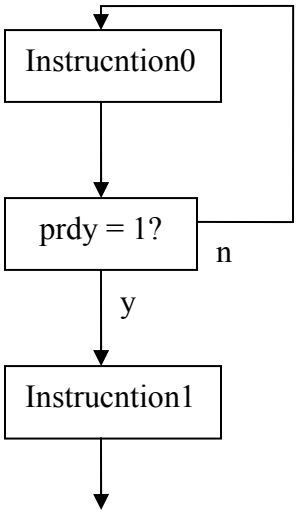


Figure 5.15 Diagram of the instruction generation

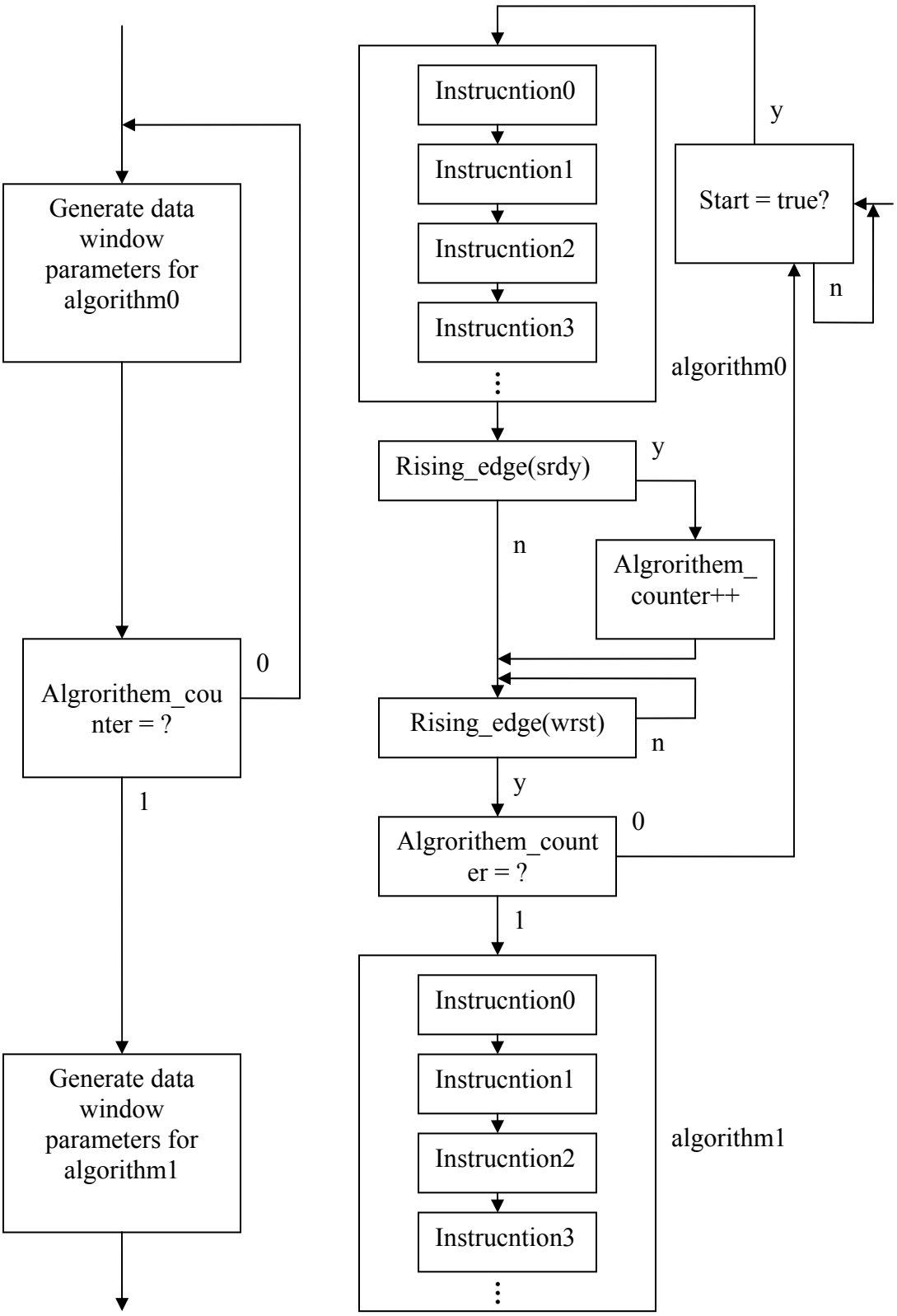


Figure 5.16 Diagram of the data stream and instruction stream generation

5.8.2 Output buffer

In SIMD controller, there is a piece of output buffer. When the processor requires writing results into external memory, a write signal is set to 1 to inform the output buffer controller. The pixel processor never writes data into external memory directly because of two reasons. First, the processors always send write requests concurrently but there is only one data bus. Second, the next data load stage is overlapped with the process stage and the data bus is probably in use when there comes a write request.

After receiving the write request, the buffer controller writes the results from 16 processors into output buffer and remembers the start address of the destination (carried in add_r signal). When the process and data load stage is finished, the controller informs that the store stage starts and writes the data in the buffer to external memory serially. However, the number of write requests is algorithm dependent. In this implementation, I choose to allocate 3×16 pieces of output buffer, each for a 32bit result. That means in a process stage, at most 3 write requests can be accepted by the buffer controller, otherwise, there will be an overflow error. This is enough for fingerprint enhancement and if applying other algorithms in this SIMD architecture, the buffer size may need adjustment according to the algorithm.

Four types of write mode are supported by the buffer controller, indicated by the13th and 14th bits in the opcode.

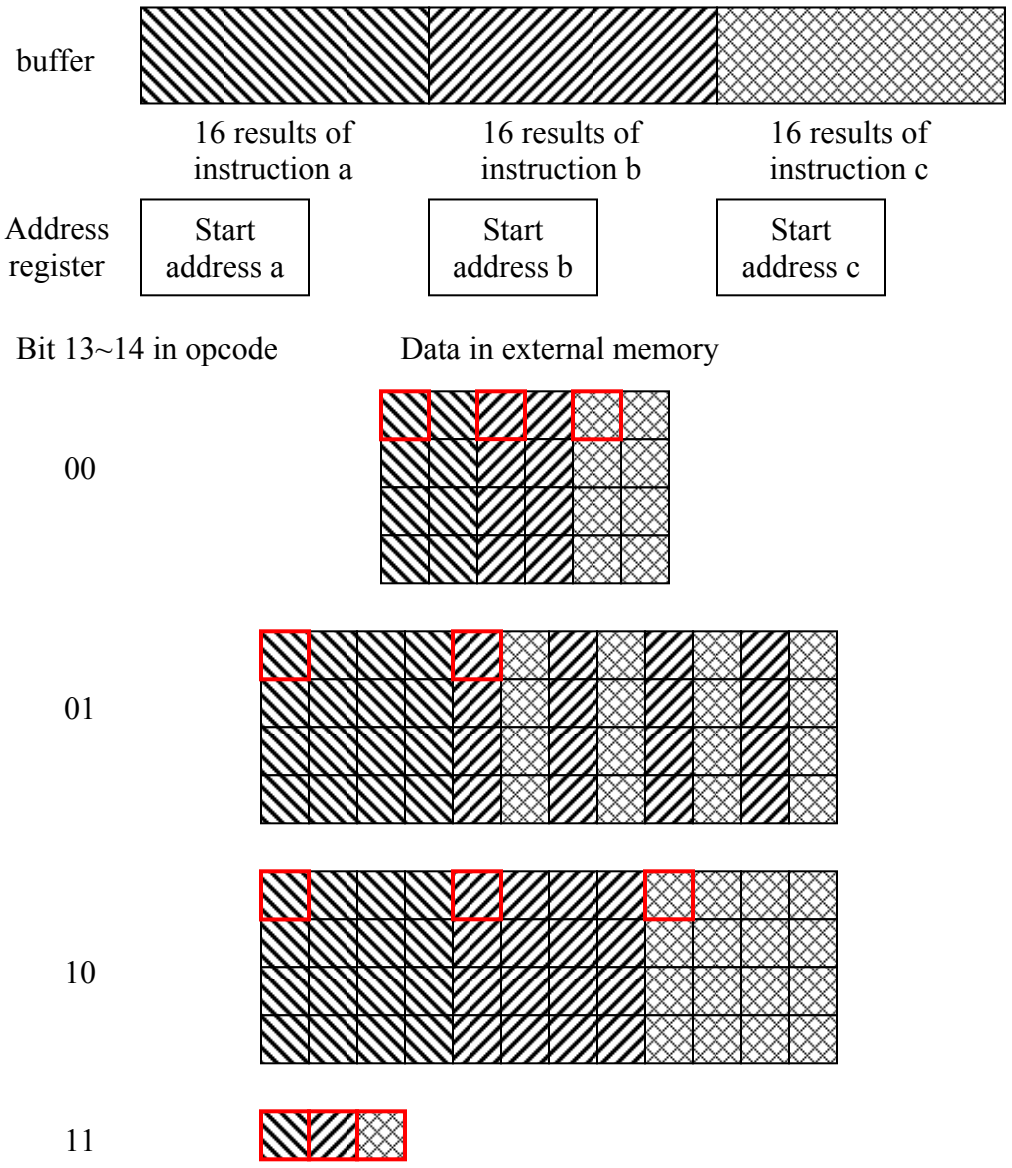


Figure 5.17 Buffer controller write back mode

Figure 5.17 shows the 4 kinds of data write mode in buffer controller. Each block denotes a 32bit wide data piece. And they exit in the external memory in the same order as the original fingerprint image, that is, from left to right in the row and from top to bottom in the column. The width of the row equals the width of the image (in pixels) times data piece width (in bytes). The data piece in red line implies that its address in the external memory equals the address in the corresponding address register.

- In the first mode, only the lower 16 bits of results are written back to the external memory.
- In the second mode, the results from two instructions are joined together in order to mix them up. And then they can be fetched within one data window.
- In the third mode, 32bit results are written back to external memory.
- In the last mode, only the 32bit result from processor 0 will be stored.

5.8.3 Sliding mask

Each sliding mask bit corresponds to a pixel array in the image. If this bit is set to 0, this pixel array will be skipped by the processor array and no algorithm is applied on it. There is no spare bit in the opcode to indicate an operation for writing to the sliding mask. Thus the first bit in the addr_r signal is used as a substitution since the address in the sliding mask never uses 32 bits. When this bit is set to 1, the processor sets w0 signal to 1 and informs SIMD controller to write the result from processor 0 to the sliding mask. This write operation is executed immediately after the rising edge of w0.

5.8.4 Ports in SIMD controller

SIMD controller is implemented in SIMD.vhd and the following table gives it in and out ports.

Name	description
clk	Input clock signal.
Rst	Input reset signal.
Rdy	Output ready signal.

Table 5.11 Ports in SIMD controller

5.9 Library

Two library files are included in this implementation, stdlib.vhd and my_package.vhd. The first one is obtain from GRLIB VHDL IP LIBRARY (Copyright (C) 2004 GAISLER RESEARCH). In this library, two functions are used to debug the VHDL program and give me a lot of convenience. They are print() and tost(). The first function can print strings on the output window in ModelSim and the second can convert other types into string type. With these two functions many intermediate results can be examined on the screen.

All the components and global types are declared in my_package.vhd. Besides, there are three important procedures:

- Init_memory: this procedure reads from the initial data from files and store in the external memory.
- Dis_memory: this procedure can writes the desired data in the external memory to files for later debug.
- Get_mask: this procedure generates the data mask for each burst operation.

The first two procedures are only used for simulation while the last will be synthesized into circuit.

5.10 Global view of the implementation

In the previous sections, we have described the implementation of all the modules in SIMD architecture. Under the control of SIMD controller, all these modules work together and provide a fast hardware solution for fingerprint enhancement with low hardware cost and power consumption. Figure 5.18 describes the structure of this implementation.

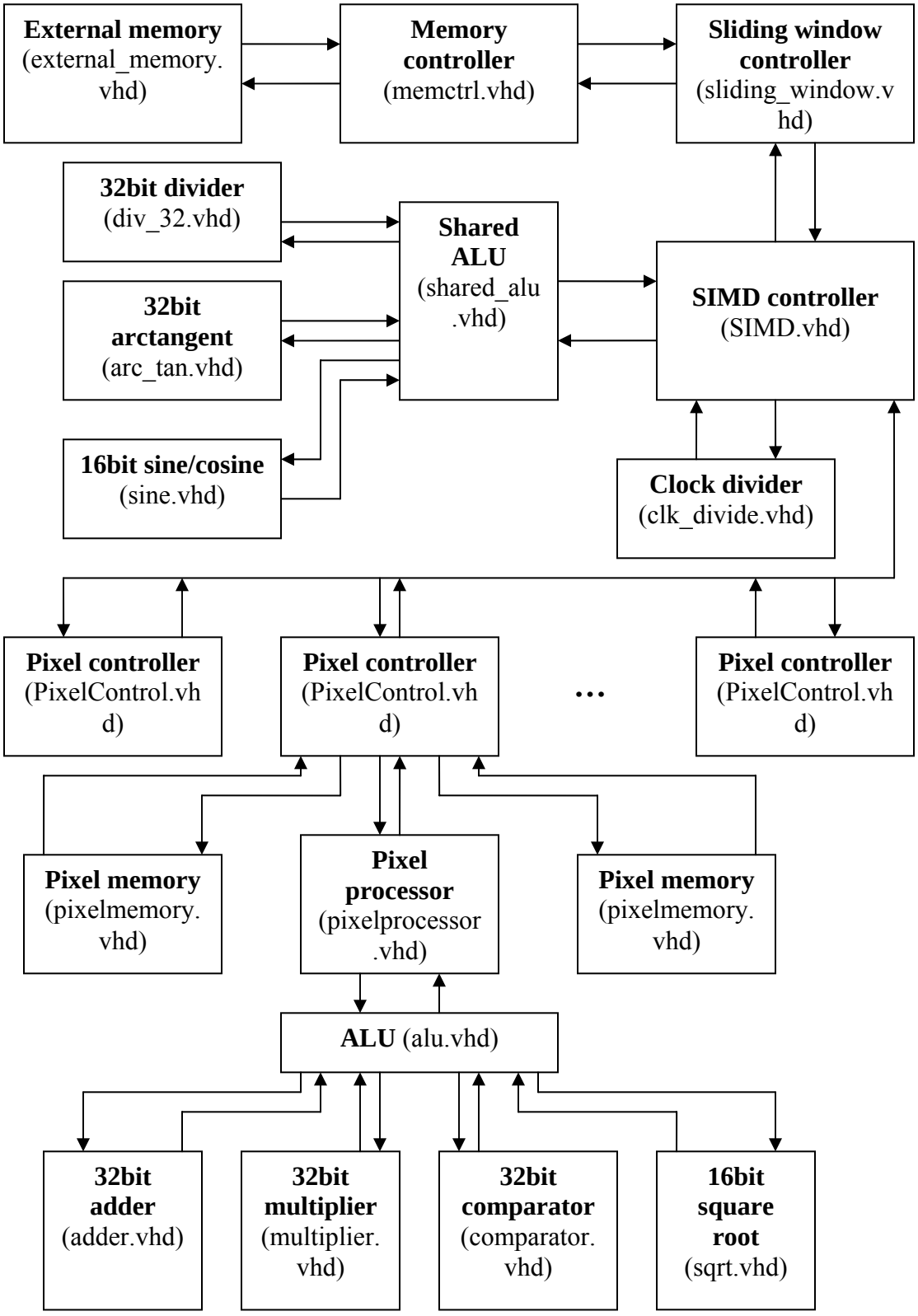


Figure 5.18 Structure of the implementation

5.11 Conclusion

In this chapter, the implementation in VHDL of the SIMD architecture is given. This architecture is originally aiming on executing the fingerprint enhancement algorithm at a high speed with low hardware cost and power consumption, which can be used in a smart card. And in the future, this architecture is potential to be extended in applying more algorithms in image processing.

Chapter 6 Simulation and evaluation

After finishing the SIMD architecture in VHDL, the rest work is divided into 3 parts: code debugging, algorithm compiling and simulation result evaluation. Each design module is test separately first and then conjointly. The algorithms in C++ are adjusted and compiled into binary codes in a format that supported by the SIMD architecture and then stored in the SIMD controller module. In the last step, the algorithms are simulated with ModelSim SE PLUS 5.8c also first separately and then conjointly come to the enhancement result. Both the time cost and enhancement quality are evaluated by comparing with the software implementation and finally it proves that our SIMD architecture meet our goal!

This chapter is organized as follows: section 6.1 presents the debugging phase for the VHDL codes and section 6.2 focuses on the code compiling for our SIMD architecture. In section 6.3 the simulation result is evaluated and the last section draws to a short conclutsion.

6.1 Code debugging

Since the architecture is quite large with a number of modules, testing separately in each module brings much convenience in finding the bugs. The results and intermediate results can be examined on the wave window from ModelSim. And the variables in the processes can also be tracked by adding test signals at the out port. Also the values of signals and variables can be printed out on the text output area with the print() function

provided in stdlib.vhd. In this way, desired signals or variables can be selected by simple programming.

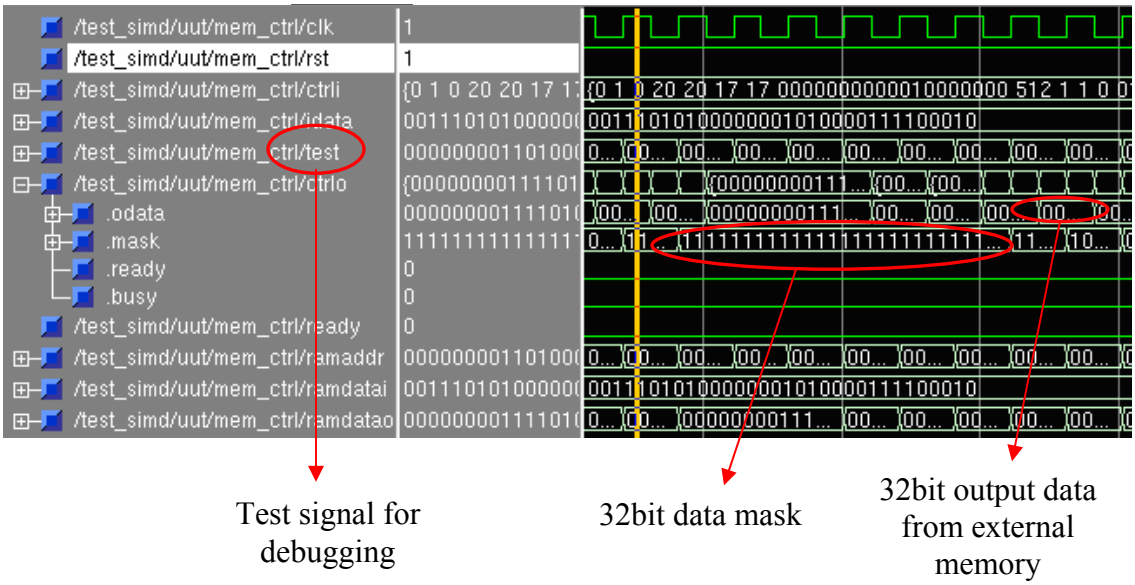


Figure 6.1 Debugging memory controller

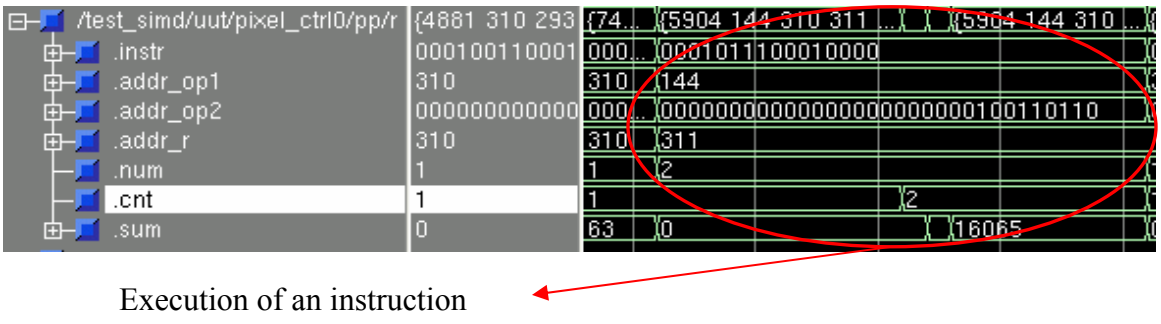


Figure 6.2 Debugging pixel processor

Figure 6.1 is a screen shot from memory controller debugging. The test signal is added for examining desired signal or variable and will be removed after the debugging. Every cycle 32bit data piece is fetched from external memory while also a 32bit data mask is generated. And in figure 6.2, the execution of an instruction is shown by the waveform. This is a multiplication instruction with operand 0 in pixel memory and operand 1 in processor cache. Thus, an extra cycle is needed to fetch operand 0 and in the second cycle the product is calculated and stored in processor cache.

For those signals can not be examined by waveform, I print them out on the text output area in the console as shown in figure 6.3. The initial image data is printed out in the right, the first integer is the address and the second denotes the 32bit data piece at that address.

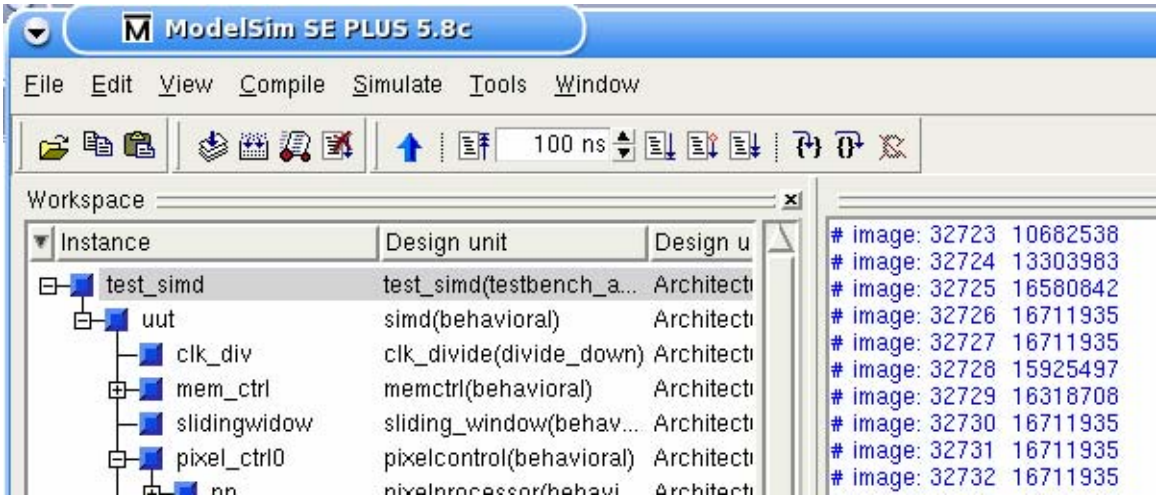


Figure 6.3 Data debugging in text form

6.2 Algorithm compiling

Unlike that in the implementation with a General-Purpose Processor the C/C++ codes can be compiled by the compiler, I have to compile all the C++ codes by hand. Due to the special architecture and instruction set, I also made slight changes in the codes while compiling. These changes mainly exist in the instruction jump at the branches. As mentioned before, if all the processors jump into the same branch which means they have the meet same condition, the instruction jump can be managed in SIMD controller and the same instruction stream is dispatched to each processor. While the instruction jump for each processor is unknown at the compile time, it has to be finished by adding idle instructions in the undesired branch at run time by processor. This also gives a solution for the synchronization among processors.

An example is given in the following. At the beginning of process stage, the SIMD controller will jump into state0, and in this state, a[i] is compared with 1. If a[i] < 1, in the next instruction cycle the processor will add an idle instruction automatically, otherwise the assignment is executed. And in state3, the loop condition is managed by SIMD controller and the processor does nothing in fact. If the loop is not finished, SIMD

controller jumps back into state 0. Otherwise it jumps into state99, where the flag is set for the end of the process stage.

C++ codes	Compiled codes

For(i=0;i<255;i++)
If(a[i] >= 1)
b[i] = a[i]*3;
	Case state is
	When state99 =>
	Ardy <= '1';
	Prst <= '0';
	When state0 =>
	Ardy <= '0';
	Prst <= '1';
	Addr_a <= conv_std_logic_vector(i,32);
	Addr_b <= (0=>'1',others=>'0');
	Addr_r <= (0=>'1',others=>'0');
	Instr <= "0001101100100001";
	Num <= 1;
	Next_state => state1;
	When state1 =>
	Addr_a <= conv_std_logic_vector(i,32);
	Addr_b <= (0=>'1',1=>'1',others=>'0');
	Addr_r <= conv_std_logic_vector(j,32);
	Instr <= "0001011100100001";
	Next_state => state2;
	When state2 =>
	Addr_a <= conv_std_logic_vector(j,32);
	Addr_b <= (others=>'0');
	Addr_r <= conv_std_logic_vector(j,32);
	Instr <= "0001001100100001";
	Num <= 1;
	i := i + 1;
	j := j + 1;
	if(i = 255) then
	Next_state => state99;
	i := 0;
	j := 0;
	else
	Next_state => state0;
	End if;
	•••

Table 6.1 Example of code compiling

6.3 Simulation evaluation

In this section the simulation result is evaluated in both time cost and enhancement quality. And we also do comparisons with software implementation.

6.3.1 Enhancement quality

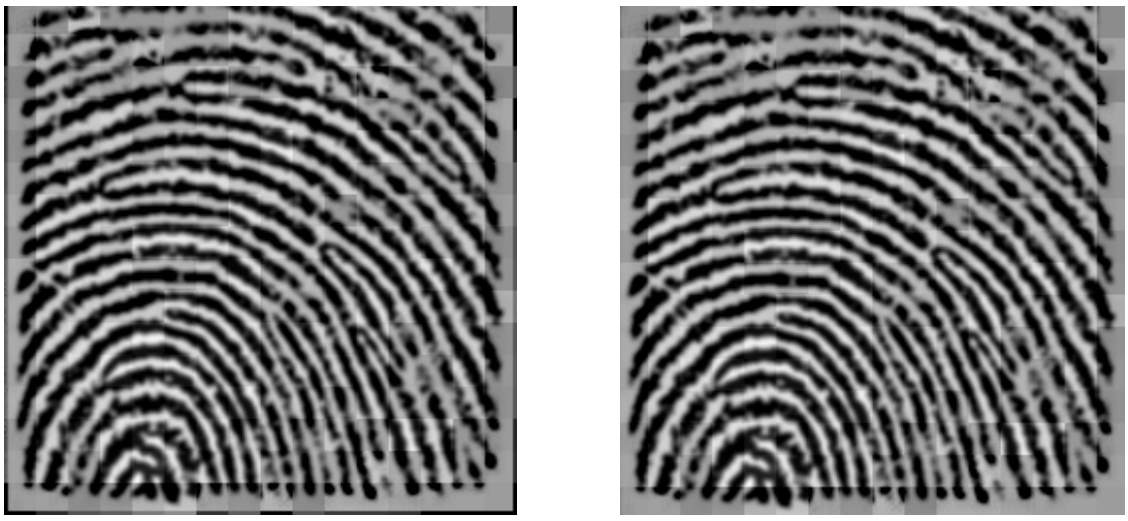
The simulation results are shown graphically in the following, comparing with those from the software implementation.

Software implementation result Hardware implementation result

Figure 6.4 Normalization result

Software implementation result Hardware implementation result

Figure 6.5 Gaussian filtering result



Software implementation result Hardware implementation result

Figure 6.6 Local normalization result

Software implementation result Hardware implementation result

Figure 6.7 Enhancement result

In figure 6.7, we can see, the enhancement result from hardware implementation is almost the same with that from software implementation. Still, there are slight differences due to the accuracy losing by fix point representation and computation error in the arithmetic units. However, these differences never introduce new spurious minutiae. This is important since it means the SIMD architecture never degrades the reliability of the enhancement algorithm.

6.3.2 Time cost

Time cost is crucial to meet the goal of real-time application. Since the VHDL codes have not been synthesized completely. I have to assume the system clock cycle as 100 MHz. This means except the memory controller and sliding window have to work at 50 MHz, the rest modules in the SIMD architecture must work at 100 MHz. Till now, the speed of the synthesized parts is listed in the following table.

Name	Speed
Memory controller	65.2 MHz
Sliding conrtoller	46.6 MHz
Pixel processor	83.7 MHz*
Pixel memory	125.6 MHz
Arithmetic units	>100 MHz
SIMD controller	Not synthesized

Table 6.2 estimated speed from synthesis

**The pixel processor is not complete synthesized and the speed is estimated by the critical path.*

From table 6.2, it's clear that more work is needed in the implementation to make the system work at 100 MHz and it also tells us it's quite possible to meet this goal. Thus we still use 100MHz as the system frequency to evaluate the time cost.

As described in figure 6.8, the total time cost is 0.752 second. Similar as in software implementation, the frequency estimation and Gabor filtering cost most of the time. In general, 0.742 second is surely enough for real-time application and the speed is boosted by 8.22 times compared with the software implementation! According to table 6.2, the possible maximum system frequency till now is 83.7 MHz, and thus the real time cost is around 0.898 second, which is also sufficient to meet the goal of real-time application.

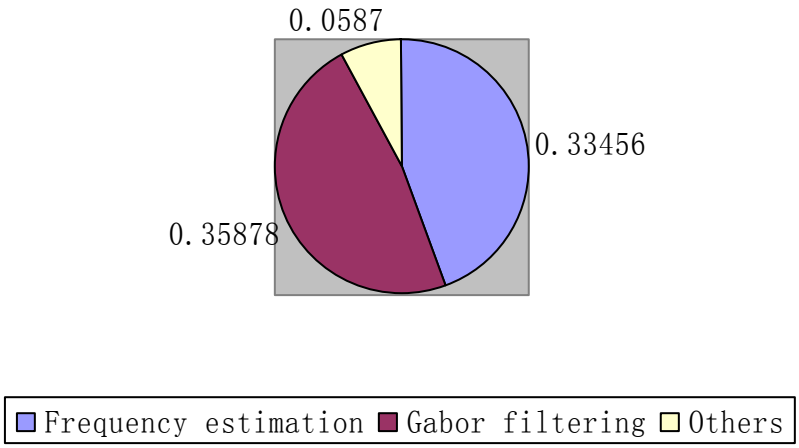


Figure 6.8 Time cost analysis at 100 MHz system frequency

6.4 Conclusion

In this chapter, the related work for hardware simulation is given and the simulation result proves that our SIMD architecture successfully meets our thesis goal both in the enhancement quality and time cost. And it is potential to enhance the system frequency for a higher speed.

Chapter 7 **Conclusions**

In the last chapter, a brief conclusion is given in section 7.1 and section 7.2 presents the future work.

7.1 **Conclusion**

In this thesis, we first propose an efficient fingerprint image enhancement algorithm based on Hong's work. This algorithm sustains the reliability of the FVS. However, due to the heavy computational complexity, it's limited in real-time application by hard constraints on hardware cost and power consumption.

In chapter 3, a software implementation is given. The simulation result shows that it's quite efficient to restore the missing minutiae information and get rid of the noise presents in the fingerprint image. And the time cost also implies that its application on embedded mobile system is seriously limited because of the computational complexity. And a hardware solution is desired for real-time application.

Then, based on the parallelism lies in the algorithm, we present a SIMD architecture. This architecture can largely increase the processing speed by exploiting the parallelism and benefits from a data pre-fetching technique. Further, fix point representation is adopted since there is no floating point units in our SIMD architecture. This brings us large benefits in hardware cost and power consumption as well as performance.

Later, the architecture is implemented in VHDL. All the algorithms are compiled into binary codes and executed in this architecture. The simulation result shows that the time cost is largely reduced to meet our goal of real-time application. And this architecture is designed aiming on running under hard constraints of hardware cost and power consumption. This makes it possible to apply our architecture on a smart card.

7.2 Future work

In the future, there are three main tasks. First, better algorithms are needed to get rid of the discontinuities presented in the fingerprint ridges which may degrade the accuracy of the frequency estimation. This can make the enhancement algorithm more reliable in case that the input fingerprint image is in poor quality.

Then more work is needed to optimize the VHDL codes in order to potentially increase the system frequency as well as reduce the hardware cost. And our final goal is to make it complete synthesizable and running it on Spartan3 XC3S 2000 chip.

Last, it’s desired to make this architecture fit for different hardware constraints and suitable for more algorithms in image processing.

Bibliography

[1] Anil Jain, Lin Hong, Sharathcha Pankanti, and Ruud Bolle, "Fingerprint Enhancement".

[2] Lin Hong, Student Member, IEEE, Yifei Wan, and Anil Jain, Fellow, IEEE, "Fingerprint Image Enhancement: Algorithm and Performance Evaluation".

[3] Raymond Thai, "Fingerprint Image Enhancement and Minutiae Extraction", The University of Western Australia, 2003.

[4] Ertugrul Saatci, Vedat Tavsanoglu, "Fingerprint Image Enhancement Using CNN Gabor-type Filters", South Bank University, London.

[5] Bob Fisher, Simon Perkins, Ashley Walker and Erik Wolfart *"*Hypermedia Image Processing Reference*", http://www.cee.hw.ac.uk/hipr/html/hipr_top.html*, Department of Artificial Intelligence University of Edinburgh, UK, 1994

[6] Lin Wang, Mo Dai, and Guohua Geng, "Fingerprint Image Segmentation by Energy of Gaussian-Hermite Moments".

[7] J. S, Chen, Y.S. Moon, K. F. Fong, "Efficient Fingerprint Image Enhancement for Mobile Embedded Systems".

[8] T. Y. Tang, Y. S. Moon, K. C. Chan, "Efficient Implementation of Fingerprint Verification for Mobile Embedded System using Fixed-point Arithmetic".

[9] Hennessy, Patterson, "Computer Architecture. A Quantitative Approach. 3rd Edition".

[10] Antonio Gentile, Jose L. Cruz-Rivera, D. Scott Wills, Leugim Bustelo, Jose J. Figueroa, Javier E. Fonseca-Camacho, Wilfredo, E. Lugo-Beauchamp, Ricardo Olivien, Marlyn Quifiones-Cerpa, Alexis H. Rivera-Rios, Iomar Vargas-Gonzales, Michelle Viera-Vera, "Real-Time Image Processing on a Focal Plane SIMD Array".

[11] "Spartan3 FPGA family Complete Data Sheet", Xilinx , Inc.

[12] Albert Zomaya, "Advanced Computer Architecture and Parallel Processing", Wiley Inc., New York, 2005.

[13] "CORDIC v3.0 Product Specification", Xilinx, Inc., 2005.