A Report on

**Finger print thinning**

**Using**

**C++ and VHDL**

BACHELOR OF TECHNOLOGY in

COMPUTER SCIENCE AND ENGINEERING

BY

|  |  |  |
| --- | --- | --- |
| PIYUSH BHUWALKA | 16JE002162 | 2017-18 |
| ROHIT PRAKASH BORSE | 16JE002594 | 2017-18 |
| SM KHOBAIB ALAM | 16JE002562 | 2017-18 |

UNDER THE GUIDANCE OF

**Dr. S. MUKHOPADHYAY**

**Associate Professor**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY

(INDIAN SCHOOL OF MINES)

DHANBAD-826004

**ACKNOWLEDGEMENT**

It gives us immense pleasure to extend our thanks to those who provided inestimable support for the completion of my work.

In the first place we would like to record our gratitude to Dr. S. Mukhopadhayay for his supervision, advice, guidance from the very early stage of this project and for patiently guiding me. We gratefully acknowledge him for his crucial contributions, which made him a backbone of this dissertation.

We are also thankful to Debashish Mukherjee for his invaluable support and encouragement.

We would like to thank our parents, for their faith in us and allowing us to be as ambitious as we wanted.

Further, We would like to thank all the people who have directly or indirectly helped us in completing this dissertation.

FINGERPRINT THINNING

Working with hardware implementation of fingerprint recognition system, a number of requirements must be fulfilled. These requirements are :-

(I) real-time

(2) portable

(3) embedded

(4) small size

(5) low power consumption.

The algorithm must be optimized when it is implemented in FPGA device due to some limitations or constraints, such as memory capacity, components, power consumptions, cost, module dependencies and development difficulties. In order to reduce difficulties during the development phase, a number of researches have been done using soft-core (soft processor). However, by using soft-core, more FPGA resources are needed as well as module dependency. Additionally, the use of translation algorithm in the soft-core increases the computational time. Another approach is the development of logical component/module for each process directly in the FPGA without using soft-core. It is more difficult, but the utilization of FPGA resources can be reduced. Moreover, since there are no dependencies with other module, the process becomes faster. This project is focused on developing a simple and fast fingerprint image-thinning algorithm so it can be implemented on embedded environment using Field Programmable Gate Array (FPGA) devices. This algorithm will be applied on pixel representation value '0'. For testing and performance evaluation, database of fingerprint from Fingerprint Verification Competition 2004 will be used. The thinning algorithm will be developed and simulated using C++ programming, as well as VHDL for the deployment into FPGA devices.

ALGORITHM USED

The following approach is described in, assumed that region point in the image have pixel value 'I' and background points have value '0'. The Zhang and Suen's method consist of successive passes of two basic steps applied to the contour points of the given region, where a contour point is any pixel with value 'I' and having at least one 8-neighbor valued '0'. With reference to the 8-neighborhood definition, the first step flags a contour point p for deletion if the following conditions are satisfied:



where N(pl) is the number of nonzero neighbors of pi; that is, N(p1) = p2 + p3 + ... + p9 and S(p I) is the number of 0-1 transition in the ordered sequence of p2, p3, ... , p8, p9. For example, N(p1) = 4 and S(p1) = 3

|  |  |  |
| --- | --- | --- |
| P9 | P2 | P3 |
| P8 | P1 | P4 |
| P7 | P6 | P5 |

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 1 |
| 1 | P1 | 0 |
| 1 | 0 | 1 |

In the second step, conditions (a) and (b) remain the same, but conditions (c) and (d) are changed to:

(c’) p2,p4.p8 = 0,

(d’)p2.p6.p8 = 0,

Step 1 is applied to every border pixel in the binary region under consideration. If one or more of the conditions (a) through (d) area violated, the value of the point in question is not changed. If all conditions area satisfied the point is flagged for deletion. It is important to be considered, that the point is not deleted until all border points have been processed. This prevents changing the structure of the data during execution of the algorithm. After step 1 has been applied to all border points, those that were flagged are deleted, changed to '0' for example. Then, step 2 is applied to the resulting data in exactly the same manner as step 1. It is clear that one iteration of the thinning algorithm consists of (1) applying step 1 to flag border points for deletion; (2) deleting the fagged points; (3) applying step 2 to flag the remaining border points for deletion; and finally (4) deleting the fagged points. This basic procedure is applied iteratively until no further points are deleted, at which time the algorithm terminates, yielding the skeleton of the region. Condition (a) is violated when contour point p1 has only one or seven 8-neighbors valued '1'. Having only one such neighbor implies that p1 is the end point of a skeleton stroke and obviously should not be deleted. If p1 had seven such neighbors and it was deleted, this would cause erosion into the region. Condition (b) is violated when it is applied to points on a stroke one pixel thick. Thus these conditions prevent disconnection of segments of a skeleton during the thinning operation. Conditions (c) and (d) are satisfied simultaneously by the following minimum set of values: p4 = '0', or p6 = '0', or (p2 = '0' and p8 = '0'). Thus with reference to .the neighborhood arrangement, a point that satisfies these conditions; as well as conditions (a) and (b), is east or south boundary point, or northwest comer point in the boundary. In either case, pI is not part of the skeleton and should be removed. Similarly, conditions (c') and (d') are satisfied simultaneously by the following minimum set of values: p2 = '0', or p8 = '0', or (p4 = '0' and p6 = '0'). These correspond to north or west boundary points, or a southeast comer point. Note that northeast comer points have p2 = '0' and p4 = '0' and thus satisfy conditions (c) and (d), as well as (c') and (d'). This is also true for southwest comer points, which have p6 = '0' and p8 = '0'. In developing algorithm there some considerations must be taken, such as : properties and constraints. In software base, these properties are : performance (accuracy and speed), complexity, size of code, size of templates, difficulty of development, dependency, and in hardware base these properties are : performance, size of block/modules, size of templates. And in implementing software algorithm into hardware base some constraints must be taking care, such as: memory, component block device, module dependency, difficulty of development, interfacing and handshaking, licensing, etc. Image acquisition and analysis from the fingerprint sensor determine the pixel width of the structure and the representation pixel value of the structure of fingerprint image as well as the image background. This characteristic must be considered when applying specific algorithm in fingerprint recognition system to have a good performance. There is a fact that ridge structure is thicker than valley structure. Average ridge width (typically 6 pixels) is thicker than average valley width (typically 4 pixels). Minutiae extraction is obtained from binary images of fingerprint. This thinner binary image will improve the performance, because it is easier for skeleton computation and it increases the speed of the process. For thinning process, Zhang and Suen's algorithm described in and is modified, this modification is presented in so it is suitable for the image with pixel value representation of '0' as a concern. Based on evaluating pixel value and the 8-neighborhood pixel, with pixel reference P1. And variable A is the number of transition from '0' to '1' clockwise from P2 ... P9 with representation of image is '0' for dark (black) and '1' for light (white) or region point is for pixel value ‘1’ and background point is ‘0’.

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 0 |
| 1 | P1 | 0 |
| 1 | 1 | 0 |

The original algorithm is modified and the process will be based on representation of image with '1' for light (white) and '0' for dark (black) or region point is for pixel value '0' and background point is '1'. Actually this process is the same with inverting the image and use the first algorithm to do thinning, of course inverting the image means more computational time. VHDL version of thinning algorithm is based on thinning algorithm adapted to pixel value as described in and. In high level language accessing image pixel value stored in matrix is easy by using index of the matrix for example P(row,col). But in hardware, this process is more difficult, usually data is stored in memory with sequential/serial access on it. For accessing several pixel value in the same time like 3x3 window, an approach present in is used. A FIFO register will delay incoming data by several bit according to the position as needed. For 8x8 image the FIF08 with 8 bit delayed is used, and for width x height a FIFO with width x height bit delayed is used. In consequence, there is border area, because the first 3x3 window is achieved after (2 x width + 2) data is sending to this block/module. For example, the first 3x3 window in 8x8 image is achieved after DI8 is sending to this module. It is happened also in the last 3x3 window, when all data has been send D63 in the cell 64, the center of the window is on D54. The target FPGA board for this design is ZestSC I with 8 Mbytes of memory and Xilinx Spartran 3 XC3S1000 FT256. As described in [6] this board uses a FPGA Xilinx Spartan-3 with up to I million gates. This approach is implemented using VHDL programming and some modules (FIFO and Memory Module) are developed based on IP Core Generator.

SEQUENTIAL IMPLEMENTATION OF THINNING ALGORITHM USING C++

**#include<bits/stdc++.h>**

**#include<fstream>**

**using namespace std;**

**int main()**

**{**

**ifstream infile;**

**int height,width,brightness;**

**infile.open("//home//khobaib//Desktop//finger1.pgm");**

**ofstream fileout("//home//khobaib//Desktop//finger2.pgm");**

**string head;**

**infile>>head;**

**infile>>width>>height>>brightness;**

**int img[height][width];**

**for(int i=0;i<height;i++)**

**{**

**for(int j=0;j<width;j++)**

**{**

**infile>>img[i][j];**

**if(img[i][j]<=255&&img[i][j]>=200)**

**img[i][j]=0;**

**else img[i][j]=255;**

**}**

**}**

**vector<pair<int,int> > v;**

**int k[]={-1,-1,0,1,1,1,0,-1};**

**int l[]={0,1,1,1,0,-1,-1,-1};**

**for(int r=0;r<500;r++)**

**{**

**for(int i=1;i<height-1;i++)**

**{**

**for(int j=1;j<width-1;j++)**

**{**

**int n=0,s=0;**

**for(int x=0;x<8;x++)**

**{**

**if(img[i+k[x]][j+l[x]]==255)**

**{**

**++n;**

**if(img[i+k[x-1]][j+l[x-1]]==0&&x>0)**

**++s;**

**}**

**}**

**if(n<=6&&n>=2&&s==1&&(img[i][j+1]==0||img[i+1][j]==0||(img[i-1][j]==0&&img[i-1][j-1]==0)))**

**v.push\_back({i,j});**

**}**

**}**

**bool flag=true,flag1=true;**

**if(v.size()==0)**

**flag=false;**

**for(int i=0;i<v.size();i++)**

**{**

**img[v[i].first][v[i].second]=0;**

**}**

**v.clear();**

**for(int i=1;i<height-1;i++)**

**{**

**for(int j=1;j<width-1;j++)**

**{**

**int n=0,s=0;**

**for(int x=0;x<8;x++)**

**{**

**if(img[i+k[x]][j+l[x]]==255)**

**{**

**++n;**

**if(img[i+k[x-1]][j+l[x-1]]==0&&x>0)**

**++s;**

**}**

**}**

**if(n<=6&&n>=2&&s==1&&(img[i-1][j-1]==0||img[i-1][j]==0||(img[i][j+1]==0&&img[i+1][j]==0)))**

**v.push\_back({i,j});**

**}**

**}**

**if(v.size()==0)**

**flag1=false;**

**for(int i=0;i<v.size();i++)**

**{**

**img[v[i].first][v[i].second]=0;**

**}**

**v.clear();**

**if(flag1==false&&flag==false)**

**break;**

**}**

**fileout<<head<<endl;**

**fileout<<width<<" "<<height<<endl;**

**fileout<<brightness<<endl;**

**for(int i=0;i<height;i++)**

**{**

**for(int j=0;j<width;j++)**

**{**

**if(img[i][j])**

**fileout<<0<<" ";**

**else**

**fileout<<255<<" ";**

**}**

**fileout<<endl;**

**}**

**infile.close();**

**return 0;**

**}**

Above c++ code takes image file as input and then sequentially determines which all pixels are to be deleted and then finally deletes them, the process is iterated till no more pixel is to be deleted.

PARALLEL IMPLEMENTATION USING VHDL

VHDL is used in this project to implement the finger print thinning algorithm. The sequential algorithm was already implemented using C++, but the main aim was to reduce time and one of the methods to achieve this is the parallel implementation of the algorithm using VHDL.

Our main VHDL module consists of 3 processors each having 4 matrices of registers of dimension 3 X (Image Width/3).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

1. The first processor contains matrices of registers which

store information about the pixels colored in black.

1. The second processor contains matrices of registers which store information about the pixels colored in red.
2. The third processor contains matrices of registers which store information about the pixels colored in yellow.

Fig 1.

The information is :-

1. Count of no. of 1
2. Count of no. of 0->1 transition
3. And product of three pixels as given in algorithm.
4. And product of other three pixels as given in algorithm.

**PROCESSOR 1**

According to Fig 1. Processor 1 contains following matrix.

Our main module is dependent on two other packages. First package if used for implementing division and modulo functions in order to make our code synthesizable. Second package is used for declaration of important data structures which are used in our main module.

Following code is used for first package :-

**library IEEE;**

**use IEEE.std\_logic\_1164.all;**

**use ieee.numeric\_std.all;**

**package division is**

**function d (a : UNSIGNED; b : UNSIGNED) return UNSIGNED is**

**variable a1 : unsigned(a'length-1 downto 0):=a;**

**variable b1 : unsigned(b'length-1 downto 0):=b;**

**variable p1 : unsigned(b'length downto 0):= (others => '0');**

**variable i : integer:=0;**

**begin**

**for i in 0 to b'length-1 loop**

**p1(b'length-1 downto 1) := p1(b'length-2 downto 0);**

**p1(0) := a1(a'length-1);**

**a1(a'length-1 downto 1) := a1(a'length-2 downto 0);**

**p1 := p1-b1;**

**if(p1(b'length-1) ='1') then**

**a1(0) :='0';**

**p1 := p1+b1;**

**else**

**a1(0) :='1';**

**end if;**

**end loop;**

**return a1;**

**end d;**

**function mo(a : integer; b : integer) return integer is**

**variable a1 : integer:=a;**

**variable b1 : integer:=0;**

**variable i: integer :=0;**

**begin**

**for i in 0 to 100 loop**

**if(b1+b <= a1 ) then**

**b1 := b1+b;**

**end if;**

**end loop;**

**return (a1-b1);**

**end mo;**

**end division;**

Following code is used for second package :-

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.all;**

**package pkg\_utility is**

**type std\_logic\_2d is array(253 downto 0,85 downto 0) of std\_logic;**

**end pkg\_utility;**

**package body pkg\_utility is**

**end pkg\_utility;**

Main Module

**library ieee;**

**use ieee.std\_logic\_1164.all;**

**use ieee.numeric\_std.ALL;**

Above code imports important libraries required.

**use work.pkg\_utility.ALL;**

**use work.division.ALL**

Importing the above mentioned packages.

**entity thin is**

**generic( N: integer := 256;**

**M: integer := 256;**

**OUT\_WIDTH: integer :=3 ;**

**KW: unsigned(1 downto 0) := "11";**

**DEPTH: integer := 86);**

**port( C: in STD\_LOGIC;**

**rst: in STD\_LOGIC;**

**din : in std\_logic;**

**Q : out std\_logic\_2d);**

**end thin;**

N is number of rows.

M is number of columns.

KW is kernel Width

DEPTH is ceiling of Image width divided by 3.

C is Clock.

rst is Reset.

din is data input.

Q is output matrix which stores whether the cell data is satisfying the four conditions used in algorithm.

**architecture bhv of thin is**

**type t\_dim2 is array(2 downto 0,DEPTH-1 downto 0) of std\_logic\_vector(OUT\_WIDTH-1 downto 0);**

**type t\_dim1\_vector\_2 is array(M-1 downto 0) of STD\_LOGIC;**

**signal temp: std\_logic\_vector(15 downto 0) := (others => '0');**

**signal t1 : std\_logic\_vector(7 downto 0) :=(others => '0');**

**signal t2 : std\_logic\_vector(7 downto 0):=(others => '0');**

**signal mtx: t\_dim2;**

**signal f\_m : std\_logic\_2d;**

**signal t\_store : t\_dim2;**

**signal t\_m\_a : t\_dim1\_vector\_2;**

**signal f\_t : std\_logic\_2d;**

**signal b : boolean;**

**signal b1 : boolean;**

**signal Q1 : std\_logic\_2d;**

temp is used for counting pixels.

t1 is used for counting columns.

t2 is used for counting rows.

mtx stores matix of how many 1’s are there surrounding to the corresponding middle pixel.

f\_m is a bool type matrix which carries the information that whether the cell satisfies the condition of numbers of 1’s around the cell is 6 or not.

t\_m\_a is used to store pixel data of one row of the matrix.

t\_store stores number of 0->1 transition around the middle pixel.

f\_t is also a bool type matrix which carries the information that whether the cell stratifies the condition of number of 0->1 transitions around the cell is 1 or not.

Q1 is the matrix which stores the information about the cell that whether it satisfies all the four conditions or not.

**begin**

**PIXEL\_COUNT: process (C)**

**begin**

**if (C'EVENT and C = '1') then**

**if rst = '1' then**

**temp <= (others => '0');**

**else**

**temp<=std\_logic\_vector(unsigned(temp)+1);**

**if unsigned(temp) = 65535 then**

**temp <= (others => '0');**

**end if;**

**end if;**

**end if;**

**end process PIXEL\_COUNT;**

The above portion of code is used for storing the number of pixel count.

**COLUMN\_COUNTER: process(C)**

**begin**

**if (C'EVENT and C = '1') then**

**if rst = '1' then**

**t1 <= (others => '0');**

**else**

**t1 <= std\_logic\_vector(unsigned(t1) +1);**

**if unsigned(t1) = M-1 then**

**t1 <= (others => '0');**

**end if;**

**end if;**

**end if;**

**end process COLUMN\_COUNTER;**

Above portion of code is used for counting column and stores it in t1.

**ROW\_COUNTER: process(C,temp,t1)**

**begin**

**if C'EVENT and C = '1' then**

**if rst = '1' then**

**t2 <= (others => '0');**

**else**

**if unsigned(t1) = 0 then**

**if unsigned(temp) > 1 then**

**t2 <= std\_logic\_vector(unsigned(t2)+1);**

**end if;**

**end if;**

**end if;**

**end if;**

**end process ROW\_COUNTER;**

Above portion of code is used for counting row and stores it in t2.

**THINNING\_PROCESS: process(t2,t1,t\_store,t\_m\_a,mtx,b,b1,din,f\_m,f\_t,Q1)**

**begin**

**if (unsigned(t2) > 2) then**

**if(to\_integer(unsigned(t\_store(mo(to\_integer(unsigned(t2)),to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))))= 1) then**

**f\_t(to\_integer(unsigned(t2)+KW)-1,to\_integer(d(unsigned(t1),KW))) <= '1';**

**end if ; t\_store(mo(to\_integer(unsigned(t2)),to\_integer(KW)),to\_integer(d(unsigned(t1),KW))) <= (others => '0');**

**end if;**

This portion of the code checks after completely storing the no. of 0->1 transition that whether it is equal to 0 to not If yes, then the corresponding cell of flag transition matrix is made true otherwise false. The cell of the store matrix is again reinitialized to zero.

**if(unsigned(t2)>0) then**

**if(din = '1') then**

**b <= true;**

**else b <= false;**

**end if;**

**if(t\_m\_a(to\_integer(unsigned(t1)))= '1') then**

**b1 <= true;**

**else b1 <= false;**

**end if;**

**if((b xor b1) and (mo(to\_integer(unsigned(t1)),to\_integer(KW)) )/= 1) then**

**t\_store(mo(((mo(to\_integer(unsigned(t2)),to\_integer(KW)))+1 ),to\_integer(KW)),to\_integer(d(unsigned(t1),KW))) <= std\_logic\_vector(unsigned(t\_store(mo(((mo(to\_integer(unsigned(t2)),to\_integer(KW)) )+1 ),to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))) + 1);**

**t\_store(mo((mo(((mo(to\_integer(unsigned(t2)),to\_integer(KW)))+1 ),to\_integer(KW)) + 1),to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))<= std\_logic\_vector(unsigned(t\_store(mo((mo(((mo(to\_integer(unsigned(t2)),to\_integer( KW)) )+1 ),to\_integer(KW))),to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))) + 1);**

**end if;**

**end if;**

If we get a 0->1 transition vertically, then the corresponding cells of store matrix is incremented by 1.

**t\_m\_a(to\_integer(unsigned(t1))) <= din;**

**if((mo(to\_integer(unsigned(t1)),to\_integer(KW))) /=0) then**

**if(din = '1' and t\_m\_a(to\_integer(unsigned(t1))-1) = '0') then**

**t\_store(mo(to\_integer(unsigned(t2)),to\_integer( KW)),to\_integer(d(unsigned(t1),KW))) <= std\_logic\_vector(unsigned(t\_store(mo(to\_integer(unsigned(t2)),to\_integer( KW)),to\_integer(d(unsigned(t1),KW)))) + 1);**

**elsif(din='0' and t\_m\_a(to\_integer(unsigned(t1))-1) = '1') then**

**t\_store(mo((to\_integer(unsigned(t2) +KW)-1),to\_integer(KW)),to\_integer(d(unsigned(t1),KW))) <= std\_logic\_vector(unsigned(t\_store(mo(to\_integer((unsigned(t2) +KW)-1),to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))) + 1);**

**end if;**

**end if;**

If we get a 0->1 transition horizontally, then the corresponding cells of store matrix is incremented by 1.

**if(din = '1') then**

In order to calculate number of 1 around the cell the conditions are checked if the input is 1 else not required.

**if(not(((mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW))) =0 and (mo((to\_integer(unsigned(t1)+KW)-1),to\_integer(KW)))=0) and (to\_integer(unsigned(t2)))<N-2))then**

**if(mo(to\_integer(unsigned(t2)),to\_integer( KW)) = 0 ) thenif(to\_integer(unsigned(mtx(mo(to\_integer(unsigned(t2)),to\_integer(KW)),to\_integer(d(unsigned(t1),KW))))) = 6 ) then**

**f\_m(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) <= '1';**

**end if;**

**Q1(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) <= f\_m(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) and f\_t(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW)));**

**mtx(mo(to\_integer(unsigned(t2)),to\_integer( KW)),to\_integer(d(unsigned(t1),KW)))<=std\_logic\_vector(to\_unsigned(0,4));**

We use only 3 rows of registers to check the no. of 1’s around any cell.

Only 3 rows are required because at an instant only three consecutive cells in a processor are going to be affected by the current input. When a 3 X 3 widow work is complete we again reinitialize the cell of the matrix to zero and then start to store the data of new upcoming cell. This way only 3 rows of registers are required. At the same time we also check whether the data present in previous window satisfies the condition and if it satisfies we flag it as 1.

**else**

**mtx(mo(to\_integer(unsigned(t2)),to\_integer( KW)),to\_integer(d(unsigned(t1),KW))) <= std\_logic\_vector(unsigned(mtx(mo(to\_integer(unsigned(t2)),to\_integer( KW)),to\_integer(d(unsigned(t1),KW)))) + 1);**

**end if;**

**end if;**

Else if we are in middle of storing the information about any window then we increment the value in the corresponding cell by 1.

**if(not(((mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW)) )=1 and (mo((to\_integer(unsigned(t1)+KW)-1),to\_integer(KW)))=0) and to\_integer(unsigned(t2))>0 and to\_integer(unsigned(t2))<N-1)) then**

**if((mo(to\_integer(unsigned(t2)),to\_integer( KW))) = 0) then if(to\_integer(unsigned(mtx(mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))))=6) then**

**f\_m(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) <= '1';**

**end if;**

**Q1(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) <= f\_m(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) and f\_t(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) ;**

**mtx(mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW)),to\_integer(d(unsigned(t1),KW)))<=std\_logic\_vector(to\_unsigned(0,4));**

**else**

**mtx(mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW)),to\_integer(d(unsigned(t1),KW))) <= std\_logic\_vector(unsigned(mtx(mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW)),to\_integer(d(unsigned(t1),KW))))+1) ;**

**end if;**

**end if;**

This piece of code checks the condition for the cell present in second row of the matrix of registers.

**if(not((mo(to\_integer(unsigned(t2)+KW)-1,to\_integer(KW)) =(to\_integer(KW)-1) and (mo(to\_integer(unsigned(t1)+KW)-1,to\_integer(KW))=0) and to\_integer(unsigned(t2))>1))) then**

**if(mo(to\_integer(unsigned(t2)),to\_integer( KW)) = 0)then**

**if(to\_integer(unsigned(mtx(mo(to\_integer(unsigned(t2)+KW)-2,to\_integer(KW)),to\_integer(d(unsigned(t1),KW))))) = 6) then**

**f\_m(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) <= '1';**

**end if;**

**Q1(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) <= f\_m(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW))) and f\_t(to\_integer(unsigned(t2))-to\_integer(KW+1),to\_integer(d(unsigned(t1),KW)));**

**mtx((mo(to\_integer(unsigned(t2)+KW+1),to\_integer(KW))),to\_integer(d(unsigned(t1),KW))) <=std\_logic\_vector(to\_unsigned(0,4));**

**else**

**mtx(mo(to\_integer(unsigned(t2)+KW)+1,to\_integer(KW)),to\_integer(d(unsigned(t1),KW))) <= std\_logic\_vector(unsigned(mtx(mo(to\_integer(unsigned(t2)+KW+1),to\_integer(KW)),to\_integer(d(unsigned(t1),KW))))+1);**

**end if;**

**end if;**

**end if;**

**end process THINNING\_PROCESS;**

This piece of code checks the condition for the cell present in third row of the matrix of registers.

**Q<=Q1;**

**end bhv;**

Similarly codes are written for processor 2 and 3 .

All the three codes are run at the same time which helps in thinning the image in less time.

After doing all the processes the result in the form of the flag matrix is given as output which is then store somewhere in the system.

The second time we take the same file and image file input. If the matrix file cell is flaged as true then we change the corresponding pixel data to ‘0’.

On third pass pixels satisfying the next set of conditions are flaged and then at the fourth pass their corresponding pixel value in the image is set to ‘0’ as well.

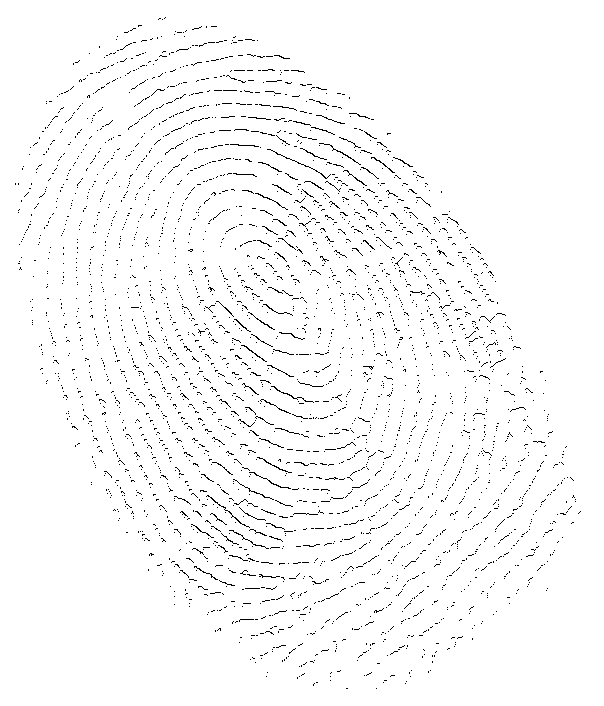
This process of 4 passes is iterated till we don’t get any pixel value which is to be deleted .

The final image obtained is the required thinned image.

RESULT



BEFORE THINNING



AFTER THINNING