Implementing a Histogram Equalization Algorithm in Reconfigurable Hardware

Stephanie Parker, Undergraduate Student, Stephanie_parker1987@yahoo.com
J. Kemi Ladeji-Osias, Associate Professor, Jumoke.Ladeji-Osias@morgan.edu
Morgan State University, Department of Electrical and Computer Engineering,
5200 Perring Parkway, Baltimore, MD 21251

Abstract

Dedicated hardware can be used when images and video are acquired and processed. In this paper, a histogram equalization algorithm is written in a Hardware Description Language for future implementation in reconfigurable hardware. The goal of this project is to implement a histogram equalization algorithm using VHDL for a real-time processing system on a Field Programmable Gate Array (FPGA). The histogram equalization algorithm was implemented and tested using a known 4 x 4 array. The array was initially coded in MATLAB and then converted to VHDL, which describes the behavior and structure of electronics systems. The algorithm was later tested using Forward Looking Infrared (FLIR) images. The simulated speed of the VHDL implementation was 0.020 ns vs. 0.053 ms in MATLAB. In conclusion, the histogram equalization has been successfully implemented in VHDL using Xilinx ISE, MATLAB and ModelSim.

Introduction

The goal of the External Hazard Detection and Classification task in NASA's Integrated Intelligent Flight Deck Project is to improve the detection of hazards due to terrain, air traffic, and runway obstacles by integrating data from weather radar, infrared video or Light Detection and Ranging (LIDAR) with existing aircraft sensors [1]. Within this project, our laboratory has focused on the use of dedicated Field Programmable Gate Array (FPGA) hardware for computationally intensive algorithms. When images and video are acquired, some manipulation and processing must occur before they are displayed. In order to maintain real-time feedback to the pilot, dedicated hardware can be used instead of software solutions.

This article evaluates a design method for a real-time processing system based on Field Programmable Gate Array (FPGA) and Digital Signal Processing (DSP) structure. To this end, a histogram equalization algorithm is implemented in reconfigurable hardware. Histogram equalization is a technique that aims to create an image with equally distributed brightness levels over the entire brightness scale [2]. The software tools that will be used are Matrix Laboratory (MATLAB), Xilinx ISE, and Mentor Graphics ModelSim. MATLAB provides the ability to write the algorithm in a high-level programming language with built-in visualization tools. Xilinx ISE and ModelSim allow for the algorithm to be written in hardware before downloading to a device. The value of this research is that hardware has a faster time in processing an image.

Section II of the paper will provide a background on histogram equalization and reconfigurable hardware. Sections III will present the methodology, while Section IV will display and discuss the results. The final two sections of the paper will provide the conclusion and future work.

Background

Image pre-processing aims to improve image data by suppressing unwilling distortions or enhancing some image features important for further processing [2]. Computers perceive images stored in arrays, while computer vision tries to duplicate the images we humans see. Digital images have a fixed number of gray-levels, based on the number of bits chosen, so gray-scale transformations are easy to realize both in hardware and software. Gray-scale transformations change the brightness of the pixels in the image and one algorithm that achieves this transformation is histogram equalization [2]. Figure 1, illustrates the steps in image processing systems applied to histogram equalization. First, the image loader is where the image is specified from either a camera or a file on the PC. Then it goes through the image processing block and inside the block, the image goes through five steps before the system outputs the equalized histogram of the original image. Once the equalized image is output to the camera, a display or one can save it to a file on the PC.

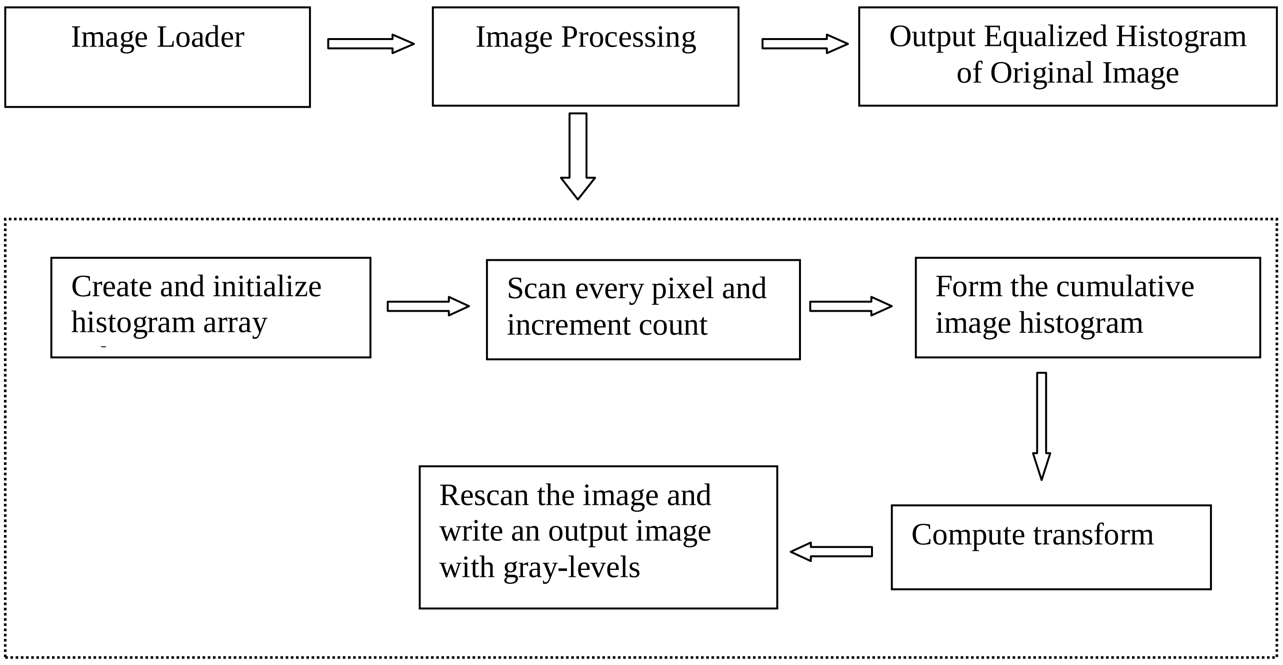


Image Loader
Image Processing
Output Equalized Histogram of Original Image
Create and initialize histogram array
Scan every pixel and increment count
Form the cumulative image histogram
Compute transform
Rescan the image and write an output image with gray-levels

Figure 1: Steps in Image Processing System

A. Histogram Equalization Algorithm

Digital images are represented as two dimensional pixel arrays. Each pixel indicates the brightness or color of the image at a given point. Histogram equalization creates an image with equally distributed brightness levels over the whole brightness scale [2]. The MATLAB high-performance language for technical computing integrates computation, visualization, and programming, and permits algorithms to be executed and simulated. MATLAB has syntax similar to many programming languages, therefore allowing one to create a code parallel to the algorithm detailed in Sonka [2]. In MATLAB, if a small set of data is used to test an algorithm, results can be extended to a larger set of data set. Although, MATLAB has a histogram equalization function (histeq) we chose to implement the algorithm using loops and lower-levels commands for portability to VHDL.

B. Reconfigurable Hardware

A FPGA is a programmable semi-conductor device that allows source code or other logic functions to be programmed in hardware description languages (HDL) to specify how the chip will work [3]. Figure 2 illustrates the steps for designing with Xilinx ISE. These include design entry, synthesis, implementation, device program and verification. These steps will ensure the necessary process to get the code to a FPGA. Xilinx ISE and ModelSim provide support for design entry using VHSIC Hardware Description Language (VHDL). VHSIC is an acronym for Very High Speed Integrated Circuit. VHDL can describe the behavior and structure of electronic systems, but is particularly suited as a language to describe the structure and behavior of digital electronic hardware designs, such as FPGAs [4].

Designs created using VHDL in Xilinx ISE allows the code to be synthesized and downloaded to the FPGA. In addition, VHDL types, functions, packages can be defined if they will be needed. ModelSim provides a comprehensive simulation and debug environment for complex FPGA designs and is accessible form ISE [5].

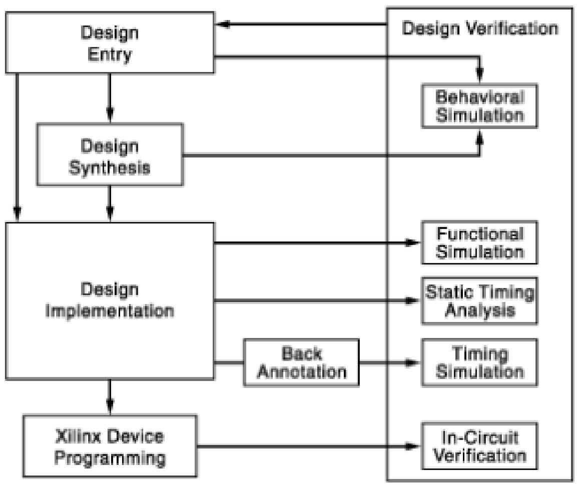


Figure 2. FPGA design flow chart [6]

Methodology

The outputs for each stage of the histogram equalization algorithm were derived for a 4x4 array. The algorithm was then implemented in MATLAB and verified using the 4x4 array and one Graphics Interchange Format (GIF) image from a previous test flight simulation. The algorithm was then coded in VHDL and tested using the 4x4 matrix.

A. Algorithm

The first step is to establish the histogram equalization algorithm that will be used. There are five steps to perform histogram equalization as described by Sonka [2].

(1) For an NxM image of G gray-levels (often 256) create an array H of length G initialized with 0 values.

(2) Form the image histogram: Scan every pixel and increment the relevant number of H—if pixel p has intensity gp, perform

$$H[gp] = H[gp] + 1 \qquad (1)$$

(3) Form the cumulative image histogram Hc:

$$\begin{aligned} Hc[0] &= H[0] \\ Hc[p] &= Hc[p-1] + H[p],\ p...2toG \end{aligned} \qquad (2)$$

(4) Set

$$T[p] = round\left(\frac{G-1}{N*M} * Hc[p]\right) \qquad (3)$$

(5) Rescan the image and write an output image with gray-levels gq, setting

$$gq = T[gp] \qquad (4)$$

B. Coding in MATLAB and VHDL

First, the values M, N, and G are defined using the 4x4 test matrix. The data values in Table 1 are defined as the 4x4 array with four gray-levels. In steps 1 and 2 the array H of the length four is initialized with zeros. A *for loop* is used to execute the code and loop back while keeping the increment index variable. In MATLAB an array index goes from one to G, therefore the algorithm is modified by adding one to gp. For step 3, a 4x1 array is defined with values of zeros. Hc[1] is assigned the value of the first element in H. Another *for loop* is applied, but this time the loop is indexed from two to G. In step 4 a mathematical expression for T[p] is executed in the *for loop*. In Step 5 the image is rescanned and the pixel is assigned the new value using T.

Once the code had the same results as the hand calculations, then the algorithm was applied to an aeronautical image by using the *imwrite* and *imread* functions to import and export the Forward Looking Infrared (FLIR) image.

The code written in MATLAB was used as a template to write the VHDL code. Xilinx ISE has a template of a VHDL module that can be customized with the inputs, outputs, and the process needed for histogram equalization. In VHDL, the array attributes were used to create the *for loops* to meet requirements for synthesis. In addition, a package was written to define the array type for the test image. The pixels in the image array were defined as a constant.

There are some differences between the two programming languages that needed to be established. For example, in MATLAB, an array index goes from one to G. In VHDL the array index ranges from 0 to G-1 as in the original algorithm. VHDL, is a type driven language and the type for each variable must be defined. VHDL defines signals which are updated differently from variables. MATLAB allows the code for Step 4 to be written in one equation. On the other hand, in VHDL, equation (3) has to be implemented using several sub steps.

VHDL does not support division. Therefore, rather than using a division operator, a shift right operation was used. The shift right function required that integers were converted to unsigned numbers first. Each time an array is shifted right, it is equivalent to divide by two. Thus, the N & M must be multiples of two;

otherwise, a divider must be designed. Once the code syntax compiled successfully, it was simulated for 100 ns, then the code's output was verified with the hand calculations and output from MATLAB.

Table 1: Original data, H (4x4 array)

1	1	2	3
1	2	2	1
3	3	1	3
2	3	1	2

Results

Hand calculations were established to make sure the code gave us the correct values for each stage of the algorithm. The column represents the array index. Table 2 (a) shows the values for each array index for step #2. The results from step #3 of the hand calculations are displayed in (b). Table 5 illustrates the values of an image with gray-levels.

Table 2: Results of Calculations for 4x4 array

(a) Histogram, H[gp] (step #2)

4	2	5	5

(b) Cumulative Histogram, Hc[p] (step #3)

4	6	11	16

(c) Transformation Matrix, T[p] (step #4)

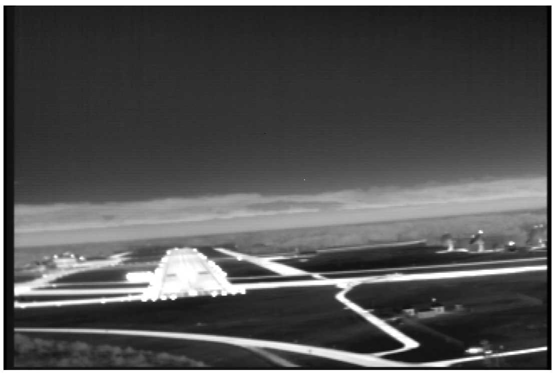
1	1	2	3

(d) New image, gp (step # 5)

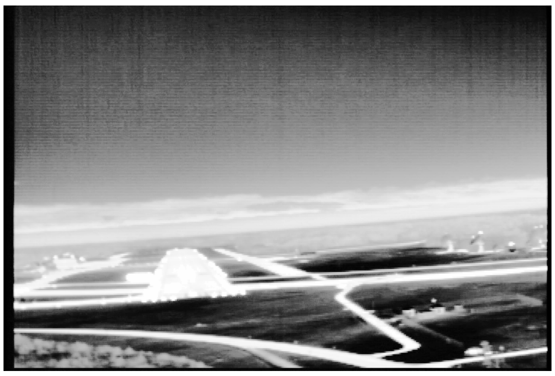
1	1	2	3
1	2	2	1
3	3	1	3
2	3	1	2

After implementation two data sets were used to evaluate the algorithm in MATLAB. The algorithm was first verified with the 4x4 matrix and hand calculations and the same result was obtained. Using FLIR images, the equalized image and the new histogram were compared to the original image. Figure 3 displays the original aeronautical and histogram equalization image. Figure 4 displays the frequency vs. intensity of the original image and histogram equalization image. The result in (b) illustrates an increase in the frequency between intensities of 50 and 250. The plot represents the graphical characteristic of the images in Figure 3. The algorithm was executed in 0.053 ms.

Upon successful implementation in MATLAB, the next step was to evaluate the algorithm in VHDL. Although the results for H and Hc[p] were valid, T was incorrect in VHDL. Table (2a) displays the incorrect values computed for T[p] after the function was implemented in the code. A second array was then used to test the code. Table 4 displays the values of the second condition, a 4x4 array, which was designed to test the code for successful executing. Table 5 are the hand calculations for the second set of data. *Shift right* (shr) function was used to implement division operator. Shifting right is dividing by two. A *if statement* was implemented to execute the rounding in Step #4. According to the least significant bit, the code will either shift right or shift right and add one to the binary number as illustrated in Figure 5. Once the code was finish and had the right syntax, then the correct values for T[p] shown in Table 5c were the same as the hand calculation. Table 5d displays the values of the equalized image. The algorithm was executed in 0.020 ms.

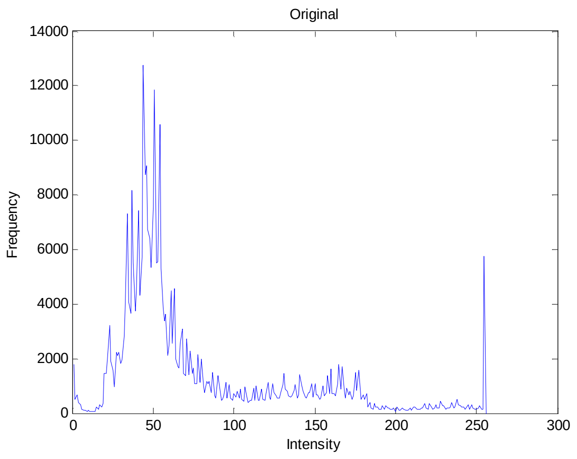


(a)

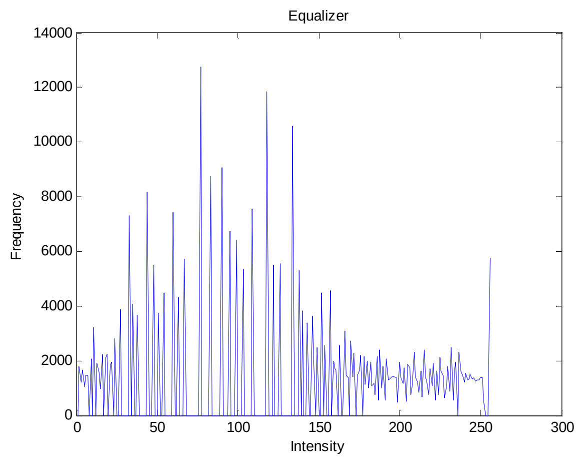


(b)

Figure 3. Histogram equalization: (a) original image; (b) equalized image.



(a)



(b)

Figure 4. Histogram equalization: (a) original histogram; (b) equalized histogram

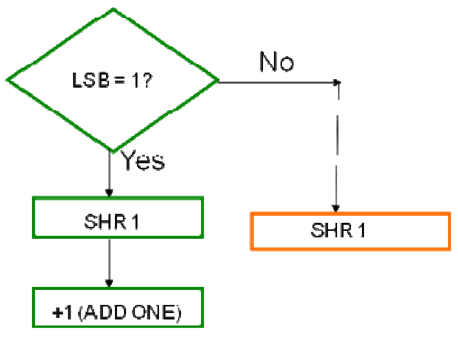


Figure 5. If statement flow chart for Rounding

Table 3: VHDL histogram equalization algorithm results

(a) T[p] (step #4)

0	1	2	3

(b) gp (step # 5)

0	1	2	3
0	2	2	0
3	3	1	3
2	3	0	2

Table 4: Second Condition (4x4 array) for testing

0	1	2	3
0	0	0	0
3	3	1	3
2	3	0	2

Table 5: Results of Calculations for Second Condition

(a) Histogram, H[p] (Step # 2)

6	2	3	5

(b) Cumulative Histogram, Hc[p] (step #3)

6	8	11	16

(c) Transformation Matrix, T[p] (step #4)

1	1	2	3

(d) New image, gp (step #5)

1	1	2	3
1	1	1	1
3	3	1	3
c2	3	1	2

Conclusion

The histogram equalization algorithm has been implemented in Xilinx ISE, MATLAB and ModelSim. In addition, preliminary synthesis has occurred. Using a small set data allowed us to write the necessary modification of division and rounding, the code will be able to perform as designed. The image dimensions must be a power of two. The code that has been written is a generic code that can be used to handle any dimensions that the user wants to utilize.

Future Research

The histogram equalization algorithm is executing correctly. The next steps will follow the FPGA design flow chart. The steps to complete this research are (1) design synthesis, (2) design implementation and (3) Xilinx device programming. Once the code is successfully downloaded on the FPGA, the next phase is to work with median filter and thresholding algorithms. After successfully implementing the algorithms in VHDL, the code will go through the rest of the FPGA design flow chart to be programmed to a FPGA.

Acknowledgement

This work was supported by the Chesapeake Information Based Aeronautics Consortium (CIBAC) at Morgan State University.

Reference

1 S.D. Young, S. Kakarlapudi, and M. Uijt de Young, "A Shadow Detection and Extraction Algorithm Using Digital Elevation models and X-band Weather Radar Measurements", *Int. J. Remote Sensing*, 26(8): 1531-1549, 2005.

2 Sonka Milan, Image Processing Analysis, and Machine Vision, 1999, California: Brooks/Cole Publishing Company

3 Field Programmable Gate Array, http://en.wikipedia.org/wiki/FPGA (Accessed 7/22/2009)

4 What is VHDL? http://74.125.47.132/search?q=cache:MlxxUJ50YEJ:www.doulos.com/knowhow/vhdl_designers_guide/what_is_vhdl/+what+is+VHDL%3F&cd=1&hl=en&ct=clnk&gl=us&client=firefox-a (Accessed 7/23/2009)

5 What is ModelSim? http://74.125.47.132/search?q=cache:r6WJWwmnOC4J:www.model.com/+what+is+ModelSim%3F&cd=3&hl=en&ct=clnk&gl=us&client=firefox-a (Accessed 7/23/2009)

6 FPGA Design Flow Overview file:///C:/Xilinx/doc/usenglish/help/iseguide/whnjs.htm (Accessed 7/27/2009)

Appendix

MATLAB CODE:

```
%Create an array H of length G initialized with 0 values
clear
tic
%initialze values for M, N and G
M=4; N=4; G = 4;
X = zeros(M,N);
X = [0 1 2 3; 0 2 2 0; 3 3 1 3; 2 3 0 2];

%Scan every pixel and increment the relevant member of H-- if pixel p has
%intensity gp, perform
H = zeros (1,G); %1x4 array
  for i = 1:M
   for j = 1:N
     gp = X(i,j);
     Temp = H(gp+1)+1;    % Adjust gp to go from 1 to G, not 0 to G-1
     H(gp+1) = Temp;
   end
  end


%Form the cumulative image histogram Hc:
Hc = zeros (1,G);
Hc(1) = H(1);
for p = 2:G
   A = Hc(p - 1) + H(p);
   Hc(p) = A;
end


%Set
T = zeros (1,G);
for p = 1:G
   A = round(((G-1)/(N*M)).*Hc(p));
   T(p) = A;
end


%Rescan the image and write an output image with gray-levels gq
T1 = zeros (M,N);
for i = 1:M
   for j = 1:N
     gp = X(i,j);
     Temp = T(gp+1);    % Adjust gp to go from 1 to G, not 0 to G-1
```

```
      T1(i,j) = Temp;
    end
end
toc
% Display output after computation is complete
H
Hc
T
T1
```

VHDL CODE:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_std.ALL;
use work.hist_pkg.ALL;
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity histo_two is
 port( --M: in integer;
   --N: in  integer;
   --G: in  integer;
   --h1: out h_vector);
  T_eq: out  hist_array);
end histo_two;
architecture Behavioral of histo_two is

--values within X
constant my_hist:hist_array:=((0,1,2,3), (0, 2, 2, 0), (3, 3, 1, 3), (2, 3, 0, 2));
----array H of length G initialized with 0 values
--type h_vector is array (0 to 3) of integer;
begin
 p0: process
 variable h: h_vector:= (0,0,0,0);
 variable gp: integer;
 variable temp: integer;

 variable H_c: h_vector;
 variable A: integer;

 variable T: h_vector;
 variable B: integer;
  variable up: integer;
  variable dwn: integer;
  variable up_u: unsigned(5 downto 0); --unsigned is array type, so have to define vector
  variable dwn_u: unsigned(5 downto 0); --unsigned is array type, so have to define vector
  variable div: unsigned(5 downto 0); --unsigned is array type, so have to define vector
  variable div_test: unsigned(5 downto 0); --unsigned is array type, so have to define vector
 variable T_eq1: hist_array;

 begin
-- Scan every pixel and increment the relevant member of H-- if pixel p has
-- intensity gp, perform
for i in my_hist'left(1) to my_hist'right(1) loop
  for j in my_hist'left(2) to my_hist'right(2) loop
  gp := my_hist(i,j);
  temp:= h(gp) + 1;
  h(gp) := temp;
  end loop;
 end loop;

-- Form the cumulative image histogram Hc:

 H_c(0) := H(0);
 for p in h'left(1)+1 to h'right(1) loop
 A := H_c(p-1) + H(p);
 H_c(p) := A;
 end loop;

 for p in h'left(1) to h'right(1) loop
   up:= (h'right(1))*(H_c(p));
    up_u:= to_UNSIGNED(up,6);
    div_test:= SHift_Right(up_u,3);--shift three times because the fourth shif depends on lsb
    if div_test(0)= '1' then
      div_test := SHift_Right(up_u,4) + 1;
    else
      div_test := Shift_Right(up_u,4);
    end if;
   B := to_INTEGER(div_test);
   T(p) := B;
 end loop;
-- Rescan the image and write an output image with gray-levels gq
 for i in my_hist'left(1) to my_hist'right(1) loop
  for j in my_hist'left(2) to my_hist'right(2) loop
  gp := my_hist(i,j);
  temp:= T(gp);
  T_eq1(i,j) := temp;
  end loop;
 end loop;
  T_eq <= T_eq1;
  --wait;
 end process;

 end Behavioral;
```