ECE 590 Final Project for Jan Fure

# Introduction

My project involved adding a median filter as an enhancement of the Erik Lindsay image processor from ECE590 year 2012. The implementation was made using a median of median, with groups of 3, for the 9 pixels that forms the image processor 3x3 kernel. Wikipedia says using a median of median algorithm; one is guaranteed that the output is between the 30th and the 70th percentile. In the case of 9 bins, running a Monte Carlo simulation of 106 points, all the points were in bins 4,5 and 6, where the median is bin 5. Using the center of each bin as percentile, the median of median filter returns outputs that are in percentiles 39,50 and 61, with the majority the 50th percentile.

Median filter was implemented the following ways:

1. Data flow behavioral programming for the Lindsay image processor, implementation of median of median.
2. Structural design of a median of medians.
3. Structural design of a median filter using horizontal sort of 3x3 matrix, followed by vertical sort, followed by diagonal sort.
4. Structural design of conventional sorter without absorption.

A Comparison of the resources required for the various implementations was performed:

MedianOfMedian: Basic building block (median3) takes three 8 bit inputs, and outputs the median. Such a block needs 3 comparators and one 3:1 mux. The MedianOfMedian component uses 4 median3 blocks, thus 12 comparators and 4 3:1 MUXs.

The sort horizontal/vertical/diagonal uses 7 sort3 blocks. A sort3 block uses 3 comparators, and three 3:1 MUXs, thus the resources expended are 21 comparators and 21 3:1 MUXs.

The conventional sorter needs 9 sort operations for 100% probability the sort of 9 elements is finished. A one million run Monte Carlo simulation shows that for nine elements, there is a 0.8% chance that the middle output of the sort is not the true median. In general, for even number of elements, the sort will finish after N-1 sort stages, whereas an odd number requires N columns of staggered minmax elements with alternating even and odd sorts. The minmax element consists of one comparator and two 2:1 MUXs. Since each column consists of one buffer plus 4 minmax elements, then 9 sort steps adds up to 36 comparators, 72 2:1 MUXs and 9 buffers. I have not yet implemented in hardware, but it seems obvious that the propagation delay is also greater for the conventional sorter then the alternative implementations.

Comparison of architectures for median

|  |  |  |  |
| --- | --- | --- | --- |
| Architecture | Conventional Sorter | Median | Median of Median |
| Details | Minmax units staggered | Horizontal/vertical/diag | 3x3 median of median |
| Comparators needed | 36 | 21 | 12 |
| MUXs needed | 72 | 21 | 4 |
| Relative delay | 9 | 3 | 2 |
| Drawbacks | Resources and delay | Resources | Accuracy |

# Median of Median Implementations

1. Integration into Liskay’s image processor:

Opcode 2A was added, which is a dataflow implementation of the median of median algorithm. Liskay’s code operates on 2D arrays of integers that are read from files on hard disk, and I cannot see a way to implement a structural element within the image processor.

1. Structural description:

|  |  |
| --- | --- |
| VHDL Code | medianofmedian3x3.vhd |
| Uses component | median3.vhd |
| Testbench | tb8.vhd |

The component median3.vhd is effectively a partial implementation of sort3.vhd, requiring 3 comparators, and one 3:1 mux.

|  |  |
| --- | --- |
| VHDL Code | median3.vhd |
| Uses behavioral components | nand3 |
|  | nand4 |
|  | nand5 |
|  | comp2in3out\_gate |
|  | mux3in1out |
| Testbench | tb2.vhd |

**Example of Median of Median for outlier removal in images**



Figure 1: Starting picture.

To test the effect on an image, a matlab function to add image noise was applied “J = imnoise(I,'salt & pepper',0.02);”, and the ability of the filter to remove the noise was tested.



Figure 2: After median of median filter.

**Discussions of the efficiency of the median of median algorithm for image processor applications.**

The median of median algorithm will return the true median of 9 inputs in 58 percent of the cases, and in all other cases, an element from inputs 4 or 6 out the sorted list of items identical with the inputs. This was verified with a java program that populated a list of 9 random numbers, reproduced the hardware median of median algorithm, and compared with all 9 random numbers sorted. One million runs result in about 580,000 cases where the return is the true median, and 180,000 cases each where the number below and above the true median gets returned. In images of real objects, the pixel values are not random, often the correlation is high. For the sake of emulating image processing, the random distribution was skewed to increase the probability of duplicate values, with increased probability of duplicate pixel values within the 9 pixels, the probability of returning the true median increases, which is what we expect. With each pixel of the last eight having a 50% probability of being a duplicate of the first pixel, the probability of returning the true median is 88%.

The test illustrated in figures 1 and 2 show that the median of median algorithm will remove outliers, and has the same low pass filter effect as a true median filter.

References:

1. <https://en.wikipedia.org/wiki/Median_of_medians>

Java code for testing:

public class Control2 {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

// TODO Auto-generated method stub

final int runs=1000000;

Data9 myData = new Data9();

for (int i=0;i<runs;i++){

myData.reGen3(0.5);

//System.out.println(myData.toString());

//System.out.println(myData.sString());

myData.MedOfMed();

}

myData.printStats();

}

}

//Argument to reGen3 is probability that a pixel of the last 8 of 9 is a duplicate of the first pixel value.

# True Median Implementation 1 (Manan’s method)

For 9 numbers, the median can be extracted using 7 sorters of 3 elements, for a 3x3 image kernel, this means sorting each row horizontally, the output of this sort gets sorted vertically, and lastly the diagonal that does not intersect the min and max gets sorted, and the median of this sort operation is the true median. From, Manan, 2011 (AKGEC Journal of Technology, January-June 2011, Vol. 2, No.1)

<http://www.akgec.in/index.php/rad/akgec-journal-of-tech/2-uncategorised/165>

http://www.akgec.in/journals/Jan-June11/V.%20(pp%2025-28)%20%20ABDUL%20Manan\_%20Implementation%20of%20Image%20Processing%20Algorithm%20on%20FPGA\_24.2.11%20%5B59%5D.doc

Link above is to online journal, and respective article. For convenience, this document is also downloaded to the folder named References.

|  |  |
| --- | --- |
| VHDL Code | median/median9.vhd |
| Uses component | median/sort3.vhd |
| Testbench | median/tb8.vhd |

# Implementation of sort3.vhd

The sort3.vhd component uses 3 comparators to determine if each pair (a,b),(b,c) and (c,a) has first input larger than, smaller than or equal to the second input. For none of the inputs equal there are 6 permutations: abc, acb, bac, bca, cab and cba. There are 3 cases of 2 inputs equal: a=b, b=c and c=a. For each of these 3 cases, the third input can be either smaller or larger than the 2 equal ones, this gives another 3x2=6 states. Lastly, all 3 inputs could be equal, a=b=c, this is the 13th state. In my implementation, I chose to give preference to the smaller indexed input when 2 are equal, meaning a>b>c outputs max\_out=a,median\_out=b,min\_out=c and the case a=b>c will have the same mux routing as the first case. This choice does not affect the final truth table or output, it only affects which mux combinations are active when two or more inputs are equal.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| In | Comp | Out | Signal | a>b>c | a=b>c | a>b=c | a=b=c | a>c>b | a=c>b | b>a>c | b>a=c | b>c>a | b=c>a | c>a>b | c>a=b | c>b>a |
| a |  | i1>i2 | C1A | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| b | COMP1 | i2>i1 | C1B | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|  |  | i1=i2 | C1EQ | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| b |  | i1>i2 | C2A | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| c | COMP2 | i2>i1 | C2B | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|  |  | i1=i2 | C2EQ | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| c |  | i1>i2 | C3A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| a | COMP3 | i2>i1 | C3B | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|  |  | i1=i2 | C3EQ | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | sort3 output ports |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | Max | a | a | a | a | a | a | b | b | b | b | c | c | c |
|  |  |  | Median | b | b | b | b | c | c | a | a | c | c | a | a | b |
|  |  |  | Min | c | c | c | c | b | b | c | c | a | a | b | b | a |
|  |  |  | State | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 | S10 | S11 | S12 | S13 |

The truth table is implemented in nand logic. Each state (S1 through S13) is detected by the nand of all high comparator outputs corresponding to the state, the corresponding signals (S1 through S13) are high for all but the detected state. The 3:1 MUXs have 3 control inputs controlling which of the 3 inputs are passed. This adds one port to the component, but greatly simplifies the logic. Now all the states where input “a” gets passed through mux are used as an input for a nand gate where high output controls the passing of “a”. The median3.vhd only uses the MUX for median, and is otherwise identical to sort3.vhd

|  |  |
| --- | --- |
| VHDL Code | sort3.vhd |
| Uses behavioral components | nand3 |
|  | nand4 |
|  | nand5 |
|  | nand6 |
|  | comp2in3out\_gate |
|  | mux3in1out |
| Testbench | tb3.vhd |

# True Median Implementation 2 (Alternating even and odd sorts, bubble sorter)

For 9 inputs, the first 4 pairs of 2 inputs are connected to minmax units, the 9th input is connected to a buffer. For the next sort step, first input goes to buffer, the next 8 go into 4 minmax units. This takes 9 operations for 100% probability that sort is completed. Since 9 inputs is an odd number, sort steps are N=9. For 8 inputs, only N-1=7 sort steps would be needed.

|  |  |
| --- | --- |
| VHDL Code | staggeredSortMedian9.vhd |
| Uses component | minmax.vhd |
| Uses behavioral component | buffer1.vhd |
| Testbench | tb6.vhd |

|  |  |
| --- | --- |
| VHDL Code | minmax.vhd |
| Uses behavioral components | mux2toOne |
|  | comp2in3out\_gate |
| Testbench | Tb5.vhd |

# Recommendations for further work in hardware image processing:

The image processor in the Liskay 2012 project

1. reads the image data from a file
2. conveniently divides it into chunks of data that can be processed in parallel
3. generates the image processor cores
4. loads the desired algorithm based on user selected opcode
5. processes the data
6. writes it to an output file.

This architecture is a great starting point for inspiration in creating a hardware image processor, but the whole architecture only allows the CASE statements in “Cell.vhd” to contain data flow statements, as they have to operate on inputs defined as arrays of integers, i.e.:

entity Cell is

generic (

IMAGE\_CHANNELS : integer := 3 -- Channels: 3=RGBColor, 1=Grayscale

);

port (

clock : in std\_logic;

opcode : in std\_logic\_vector(7 downto 0);

userInputA : in std\_logic\_vector(7 downto 0);

userInputB : in std\_logic\_vector(7 downto 0);

userKernel : in kernel;

inputA : in pixel\_array(2 downto 0,2 downto 0);

inputB : in pixel\_array(2 downto 0,2 downto 0);

outputC : out pixel\_array(2 downto 0,2 downto 0)

);

end Cell;

The code for adding to the blue intensity is:

when x"04" => -- ADDIB

for y in 0 to 2 loop

for x in 0 to 2 loop

outputC(x,y)(0) <= inputA(x,y)(0);

outputC(x,y)(1) <= inputA(x,y)(1);

outputC(x,y)(2) <= inputA(x,y)(2) + userInputA;

end loop;

end loop;

I don’t know how to add a structural component for median to this code without totally rewriting the whole image processor, but if I could, the project would still not be synthesizable, as the image comes from a file.

My recommendation is taking the existing architecture as inspiration, probably the generate statements that creates arrays of image processors can be re used, but the project has to be started from scratch, and customized to a specific FPGA hardware architecture.

Using the Atlys board

<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS>

($230)

There are 2 convenient way to source image data:

1. Use onboard HDMI port. (cheapest)
2. Use stereo camera connected to onboard VMOD connector (could be used for robot 3d vision)

<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,648,931&Prod=VMOD-CAM>

($87 + $25 for cable)

In summary, I have added a median filter, that can be used for rejecting image noise, median of median is implemented both in dataflow and structural VHDL, additionally, in structural VHDL, I have implemented a bubble sorter, and also a true median sorter architecture for 9 elements, that sorts rows, then columns, lastly the cross diagonal. If a median of median is sufficient, then the number of comparators needed is 1/3 the number required for a bubble sorter for 9 elements. These are good building blocks for later image processing applications.

The document “Howto use VHDL image processor.docx” is a quick get started guide if anybody want to get the Liskay image processor running. It took me some effort, as the input file was missing from the prior project directory, so I had to reverse engineer the format. The directory “ImageProcessor” is ready to go in ModelSim student edition or QuestaSim.

Appendix A

Code added to data flow architecture “cell.vhd” from the Liskay project:

-- INSERT MORE INSTRUCTIONS HERE!

when x"2A" => --median per channel

tempsignal\_red(7 downto 0) := inputA(0,0)(0);

tempsignal\_green(7 downto 0) := inputA(0,0)(1);

tempsignal\_blue(7 downto 0) := inputA(0,0)(2);

for x in 0 to 2 loop --red

if (inputA(x,0)(0) <= inputA(x,1)(0)) and (inputA(x,1)(0) <= inputA(x,2)(0)) then

temp\_inputA(x,0)(0) := inputA(x,1)(0);

elsif (inputA(x,2)(0) <= inputA(x,1)(0)) and (inputA(x,1)(0) <= inputA(x,0)(0)) then

temp\_inputA(x,0)(0) := inputA(x,1)(0);

elsif (inputA(x,1)(0) <= inputA(x,0)(0)) and (inputA(x,0)(0) <= inputA(x,2)(0)) then

temp\_inputA(x,0)(0) := inputA(x,0)(0);

elsif (inputA(x,2)(0) <= inputA(x,0)(0)) and (inputA(x,0)(0) <= inputA(x,1)(0)) then

temp\_inputA(x,0)(0) := inputA(x,0)(0);

elsif (inputA(x,0)(0) <= inputA(x,2)(0)) and (inputA(x,2)(0) <= inputA(x,1)(0)) then

temp\_inputA(x,0)(0) := inputA(x,2)(0);

elsif (inputA(x,1)(0) <= inputA(x,2)(0)) and (inputA(x,2)(0) <= inputA(x,0)(0)) then

temp\_inputA(x,0)(0) := inputA(x,2)(0);

else

temp\_inputA(x,0)(0) := inputA(x,1)(0);

end if;

end loop; -- x

for x in 0 to 2 loop -- green

if (inputA(x,0)(1) <= inputA(x,1)(1)) and (inputA(x,1)(1) <= inputA(x,2)(1)) then

temp\_inputA(x,0)(1) := inputA(x,1)(1);

elsif (inputA(x,2)(1) <= inputA(x,1)(1)) and (inputA(x,1)(1) <= inputA(x,0)(1)) then

temp\_inputA(x,0)(1) := inputA(x,1)(1);

elsif (inputA(x,1)(1) <= inputA(x,0)(1)) and (inputA(x,0)(1) <= inputA(x,2)(1)) then

temp\_inputA(x,0)(1) := inputA(x,0)(1);

elsif (inputA(x,2)(1) <= inputA(x,0)(1)) and (inputA(x,0)(1) <= inputA(x,1)(1)) then

temp\_inputA(x,0)(1) := inputA(x,0)(1);

elsif (inputA(x,0)(1) <= inputA(x,2)(1)) and (inputA(x,2)(1) <= inputA(x,1)(1)) then

temp\_inputA(x,0)(1) := inputA(x,2)(1);

elsif (inputA(x,1)(1) <= inputA(x,2)(1)) and (inputA(x,2)(1) <= inputA(x,0)(1)) then

temp\_inputA(x,0)(1) := inputA(x,2)(1);

else

temp\_inputA(x,0)(1) := inputA(x,1)(1);

end if;

end loop; -- x

for x in 0 to 2 loop -- blue

if (inputA(x,0)(2) <= inputA(x,1)(2)) and (inputA(x,1)(2) <= inputA(x,2)(2)) then

temp\_inputA(x,0)(2) := inputA(x,1)(2);

elsif (inputA(x,2)(2) <= inputA(x,1)(2)) and (inputA(x,1)(2) <= inputA(x,0)(2)) then

temp\_inputA(x,0)(2) := inputA(x,1)(2);

elsif (inputA(x,1)(2) <= inputA(x,0)(2)) and (inputA(x,0)(2) <= inputA(x,2)(2)) then

temp\_inputA(x,0)(2) := inputA(x,0)(2);

elsif (inputA(x,2)(2) <= inputA(x,0)(2)) and (inputA(x,0)(2) <= inputA(x,1)(2)) then

temp\_inputA(x,0)(2) := inputA(x,0)(2);

elsif (inputA(x,0)(2) <= inputA(x,2)(2)) and (inputA(x,2)(2) <= inputA(x,1)(2)) then

temp\_inputA(x,0)(2) := inputA(x,2)(2);

elsif (inputA(x,1)(2) <= inputA(x,2)(2)) and (inputA(x,2)(2) <= inputA(x,0)(2)) then

temp\_inputA(x,0)(2) := inputA(x,2)(2);

else

temp\_inputA(x,0)(2) := inputA(x,1)(2);

end if;

end loop; -- x

if (temp\_inputA(1,0)(0) <= temp\_inputA(0,0)(0)) and (temp\_inputA(0,0)(0) <= temp\_inputA(2,0)(0)) then

tempsignal\_red(7 downto 0) := temp\_inputA(0,0)(0);

elsif (temp\_inputA(2,0)(0) <= temp\_inputA(0,0)(0)) and (temp\_inputA(0,0)(0) <= temp\_inputA(1,0)(0)) then

tempsignal\_red(7 downto 0) := temp\_inputA(0,0)(0);

elsif (temp\_inputA(0,0)(0) <= temp\_inputA(1,0)(0)) and (temp\_inputA(1,0)(0) <= temp\_inputA(2,0)(0)) then

tempsignal\_red(7 downto 0) := temp\_inputA(1,0)(0);

elsif (temp\_inputA(2,0)(0) <= temp\_inputA(1,0)(0)) and (temp\_inputA(1,0)(0) <= temp\_inputA(0,0)(0)) then

tempsignal\_red(7 downto 0) := temp\_inputA(1,0)(0);

elsif (temp\_inputA(0,0)(0) <= temp\_inputA(2,0)(0)) and (temp\_inputA(2,0)(0) <= temp\_inputA(1,0)(0)) then

tempsignal\_red(7 downto 0) := temp\_inputA(2,0)(0);

elsif (temp\_inputA(1,0)(0) <= temp\_inputA(2,0)(0)) and (temp\_inputA(2,0)(0) <= temp\_inputA(0,0)(0)) then

tempsignal\_red(7 downto 0) := temp\_inputA(2,0)(0);

else

tempsignal\_red(7 downto 0) := temp\_inputA(1,0)(0);

end if;

if (temp\_inputA(1,0)(1) <= temp\_inputA(0,0)(1)) and (temp\_inputA(0,0)(1) <= temp\_inputA(2,0)(1)) then

tempsignal\_green(7 downto 0) := temp\_inputA(0,0)(1);

elsif (temp\_inputA(2,0)(1) <= temp\_inputA(0,0)(1)) and (temp\_inputA(0,0)(1) <= temp\_inputA(1,0)(1)) then

tempsignal\_green(7 downto 0) := temp\_inputA(0,0)(1);

elsif (temp\_inputA(0,0)(1) <= temp\_inputA(1,0)(1)) and (temp\_inputA(1,0)(1) <= temp\_inputA(2,0)(1)) then

tempsignal\_green(7 downto 0) := temp\_inputA(1,0)(1);

elsif (temp\_inputA(2,0)(1) <= temp\_inputA(1,0)(1)) and (temp\_inputA(1,0)(1) <= temp\_inputA(0,0)(1)) then

tempsignal\_green(7 downto 0) := temp\_inputA(1,0)(1);

elsif (temp\_inputA(0,0)(1) <= temp\_inputA(2,0)(1)) and (temp\_inputA(2,0)(1) <= temp\_inputA(1,0)(1)) then

tempsignal\_green(7 downto 0) := temp\_inputA(2,0)(1);

elsif (temp\_inputA(1,0)(1) <= temp\_inputA(2,0)(1)) and (temp\_inputA(2,0)(1) <= temp\_inputA(0,0)(1)) then

tempsignal\_green(7 downto 0) := temp\_inputA(2,0)(1);

else

tempsignal\_green(7 downto 0) := temp\_inputA(1,0)(1);

end if;

if (temp\_inputA(1,0)(2) <= temp\_inputA(0,0)(2)) and (temp\_inputA(0,0)(2) <= temp\_inputA(2,0)(2)) then

tempsignal\_blue(7 downto 0) := temp\_inputA(0,0)(2);

elsif (temp\_inputA(2,0)(2) <= temp\_inputA(0,0)(2)) and (temp\_inputA(0,0)(2) <= temp\_inputA(1,0)(2)) then

tempsignal\_blue(7 downto 0) := temp\_inputA(0,0)(2);

elsif (temp\_inputA(0,0)(2) <= temp\_inputA(1,0)(2)) and (temp\_inputA(1,0)(2) <= temp\_inputA(2,0)(2)) then

tempsignal\_blue(7 downto 0) := temp\_inputA(1,0)(2);

elsif (temp\_inputA(2,0)(2) <= temp\_inputA(1,0)(2)) and (temp\_inputA(1,0)(2) <= temp\_inputA(0,0)(2)) then

tempsignal\_blue(7 downto 0) := temp\_inputA(1,0)(2);

elsif (temp\_inputA(0,0)(2) <= temp\_inputA(2,0)(2)) and (temp\_inputA(2,0)(2) <= temp\_inputA(1,0)(2)) then

tempsignal\_blue(7 downto 0) := temp\_inputA(2,0)(2);

elsif (temp\_inputA(1,0)(2) <= temp\_inputA(2,0)(2)) and (temp\_inputA(2,0)(2) <= temp\_inputA(0,0)(2)) then

tempsignal\_blue(7 downto 0) := temp\_inputA(2,0)(2);

else

tempsignal\_blue(7 downto 0) := temp\_inputA(1,0)(2);

end if;

outputC(1,1)(0) <= tempsignal\_red(7 downto 0);

outputC(1,1)(1) <= tempsignal\_green(7 downto 0);

outputC(1,1)(2) <= tempsignal\_blue(7 downto 0);

Appendix B: sort3.vhd

library IEEE;

use IEEE.std\_logic\_1164.all;

entity sort3 is

port(

a: in STD\_LOGIC\_VECTOR(7 downto 0);

b: in STD\_LOGIC\_VECTOR(7 downto 0);

c: in STD\_LOGIC\_VECTOR(7 downto 0);

min: out STD\_LOGIC\_VECTOR(7 downto 0);

median: out STD\_LOGIC\_VECTOR(7 downto 0);

max: out STD\_LOGIC\_VECTOR(7 downto 0)

);

end sort3;

architecture s3 of sort3 is

component mux3in1out is

port(

a: in STD\_LOGIC\_VECTOR(7 downto 0);

b: in STD\_LOGIC\_VECTOR(7 downto 0);

c: in STD\_LOGIC\_VECTOR(7 downto 0);

sa: in STD\_LOGIC;

sb: in STD\_LOGIC;

sc: in STD\_LOGIC;

q: out STD\_LOGIC\_VECTOR(7 downto 0));

end component;

component comp2in3out\_gate is

port(

a: in STD\_LOGIC\_VECTOR(7 downto 0);

b: in STD\_LOGIC\_VECTOR(7 downto 0);

q1\_a: out STD\_LOGIC;

q2\_b: out STD\_LOGIC;

q3\_equal: out STD\_LOGIC);

end component;

component nand3 is

port(

a: in STD\_LOGIC;

b: in STD\_LOGIC;

c: in STD\_LOGIC;

q: out STD\_LOGIC);

end component;

component nand4 is

port(

a: in STD\_LOGIC;

b: in STD\_LOGIC;

c: in STD\_LOGIC;

d: in STD\_LOGIC;

q: out STD\_LOGIC);

end component;

component nand5 is

port(

a: in STD\_LOGIC;

b: in STD\_LOGIC;

c: in STD\_LOGIC;

d: in STD\_LOGIC;

e: in STD\_LOGIC;

q: out STD\_LOGIC);

end component;

component nand6 is

port(

a: in STD\_LOGIC;

b: in STD\_LOGIC;

c: in STD\_LOGIC;

d: in STD\_LOGIC;

e: in STD\_LOGIC;

f: in STD\_LOGIC;

q: out STD\_LOGIC);

end component;

signal C1A,C1B,C1EQ,C2A,C2B,C2EQ,C3A,C3B,C3EQ: STD\_LOGIC;

signal a\_med\_rx,b\_med\_rx,c\_med\_rx,a\_min\_rx,b\_min\_rx,c\_min\_rx,a\_max\_rx,b\_max\_rx,c\_max\_rx: STD\_LOGIC;

signaL s01,s02,s03,s04,s05,s06,s07,s08,s09,s10,s11,s12,s13: STD\_LOGIC;

signal pre\_out: STD\_LOGIC\_VECTOR(7 downto 0);

begin

MAX\_MUX: mux3in1out port map(a,b,c,a\_max\_rx,b\_max\_rx,c\_max\_rx,max);

MEDIAN\_MUX: mux3in1out port map(a,b,c,a\_med\_rx,b\_med\_rx,c\_med\_rx,median);

MIN\_MUX: mux3in1out port map(a,b,c,a\_min\_rx,b\_min\_rx,c\_min\_rx,min);

C1: comp2in3out\_gate port map(a,b,C1A,C1B,C1EQ);

C2: comp2in3out\_gate port map(b,c,C2A,C2B,C2EQ);

C3: comp2in3out\_gate port map(c,a,C3A,C3B,C3EQ);

ND1: nand3 port map(C1A,C2A,C3B,s01);

ND2: nand3 port map(C1EQ,C2A,C3B,s02);

ND3: nand3 port map(C1A,C2EQ,C3B,s03);

ND4: nand3 port map(C1EQ,C2EQ,C3EQ,s04);

ND5: nand3 port map(C1A,C2B,C3B,s05);

ND6: nand3 port map(C1A,C2B,C3EQ,s06);

ND7: nand3 port map(C1B,C2A,C3B,s07);

ND8: nand3 port map(C1B,C2A,C3EQ,s08);

ND9: nand3 port map(C1B,C2A,C3A,s09);

ND10: nand3 port map(C1B,C2EQ,C3A,s10);

ND11: nand3 port map(C1A,C2B,C3A,s11);

ND12: nand3 port map(C1EQ,C2B,C3A,s12);

ND13: nand3 port map(C1B,C2B,C3A,s13);

ND\_A\_MED: nand4 port map(s07,s08,s11,s12,a\_med\_rx);

ND\_B\_MED: nand5 port map(s01,s02,s03,s04,s13,b\_med\_rx);

ND\_C\_MED: nand4 port map(s05,s06,s09,s10,c\_med\_rx);

ND\_A\_MAX: nand6 port map(s01,s02,s03,s04,s05,s06,a\_max\_rx);

ND\_B\_MAX: nand4 port map(s07,s08,s09,s10,b\_max\_rx);

ND\_C\_MAX: nand3 port map(s11,s12,s13,c\_max\_rx);

ND\_A\_MIN: nand3 port map(s09,s10,s13,a\_min\_rx);

ND\_B\_MIN: nand4 port map(s05,s06,s11,s12,b\_min\_rx);

ND\_C\_MIN: nand6 port map(s01,s02,s03,s04,s07,s08,c\_min\_rx);

end s3;