SOBEL EDGE DETECTION ALGORITHM

INTRODUCTION

This manual contains the steps required for the implementation of the Sobel Edge Detection algorithm using VHDL, proposed on chapter 11 of "VHDL Design Representation and Synthesis". ISE 8.2 and MODEL SIM are the tools provided by Xilinx for this purpose.
The files that we will use are specified below.
Edge_detector.vhdl
Image_processing.vhdl
Test.vhdl

To prove the correctness of the implementation of the algorithm we can use the following test images. It is necessary to specify the name and the size of the image that we will use.

Im1.txt	*size 15x15*
Im2.txt	*size 30x30*
Im3.txt	*size 50x50*
Im4.txt	*size 50x50*

In addition, a MATLAB file is included in this tutorial (*convert_to_txt.m*), which can read an image in 'jpg' format (the file format can only be modified by changing the extension in the code) and store its contents in 'txt' format, the one used in the VHDL code. The provided MATLAB code can also plot the original image with the image obtained as a result of running the Sobel edge detection algorithm.

Note: the files must be in the current directory of MATLAB, so it is recommended to change the directory to the one that you created for your new project.

THEORY OF THE SOBEL EDGE DETECTION AND ITS IMPLEMENTATION IN VHDL

In this part, the sobel edge detection algorithm is described, while explaining the VHDL code used for its implementation.

In general, an algorithm of edge detection finds the sharp intensity variation of an image and in this way it obtains the edges of the objects contained on the image. There are various methods to detect the edges which use descrete gradients, laplacians, etc. The most common methods used in the detection of edges are Roberts, Sobel, Preview, Laplacian, Canny, etc. Their operators are masks of 3x3 windows (2x2 windows in the Roberts algorithm) which are convolved with the incoming image to assign each pixel a value of 0 or 255. To obtain better results each method applies between two and four masks to find edges in the image.

This tutorial explains the implementation of one of the most commonly used methods for edge detection called Sobel edge detection algorithm. This algorithm uses 4 operators (also called masks or kernels) of 3x3 windows which measure the intensity

variation of the image when they are convolved with it in 4 directions: horizontal, vertical, right diagonal and left diagonal.

Now we describe the Sobel Edge Detection algorithm and its VHDL implementation. Consider an input image of size mxn as shown in figure 1.

A(1,1)	A(1,2)	A(1,3)	…	A(1,n)
A(2,1)	A(2,2)	A(2,3)		
A(3,1)	A(3,2)	A(3,3)		
⋮	⋮	⋮		⋮
A(m,1)			…	A(m,n)

Figure 1 Arrangement of pixels for an image.

Since the masks are of dimension 3x3, it is necessary to select a 3x3 window of the image to convolve it with each mask. To explain this process we select the first 3x3 window of the image as shown in the shadowed part of the previous figure. This algorithm uses four masks as shown below, we can observe in the equations how this masks are applied to the original image.

H=

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

V=

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

DL=

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

DR=

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

To convolve the image with these masks, we have to solve some equations, which are described in the following paragraphs. Note that these equations are implemented in the package of image processing as functions which are specified for each equation.

All the equations that are shown below have the same form. The function that implemented these operations is called *weight* and the code is described as follows.

```
---------------- Weight function --------------------------------
function weight
   ( X1,X2,X3: pixel)
    return filter_out is
begin
    return X1+ 2*X2 + X3;
end weight;
```

To obtain the value of the intensity variation, the implementation in VHDL is shown for each case. Note that these functions call the weight function.

$$Eh = (A(1,1) + 2A(1,2) + A(1,3)) - (A(3,3) + 2A(3,2) + A(3,1))$$

```
---------------- Horizontal_filter function ---------------------
function horizontal_filter
    ( A: pixel3_3)
    return filter_out is
begin
   return  weight( A(1,1), A(1,2), A(1,3) )
          -weight( A(3,1), A(3,2), A(3,3) );
end horizontal_filter;
```

$$Ev = (A(1,3) + 2A(2,3) + A(3,3)) - (A(1,1) + 2A(2,1) + A(3,1))$$

```
---------------- Vertical_filter function ----------------------
function vertical_filter
    ( A: pixel3_3)
    return filter_out is
begin
     return weight( A(1,3), A(2,3), A(3,3))
           -weight(A(1,1), A(2,1), A(3,1));
end vertical_filter;
```

$$Edl = (A(1,2) + 2A(1,3) + A(2,3)) - (A(2,1) + 2A(3,1) + A(3,2))$$

```
---------------- Diagonal_l_filter function --------------------
function diagonal_l_filter
    ( A:pixel3_3)
    return filter_out is
begin
    return weight( A(1,2), A(1,3), A(2,3))
           -weight(A(2,1), A(3,1), A(3,2));
end diagonal_l_filter;
```

$$Edr = (A(1,2) + 2A(1,1) + A(2,1)) - (A(2,3) + 2A(3,3) + A(3,2))$$

```
---------------- Diagonal_r_filter function --------------------
function diagonal_r_filter
    ( A:pixel3_3)
    return filter_out is
begin
    return  weight( A(1,2), A(1,1), A(2,1) )
      -weight( A(2,3), A(3,3), A(3,2) );
end diagonal_r_filter;
```

After finding the values in each direction with each function, the gradient magnitude and the direction of the intensity variation must be computed. The formula that is used for this purpose is the following.

$$Mag = \max[|Eh|,|Ev|,|Edl|,|Edr|] + \frac{1}{8}\left\lfloor |E_{\perp}| \right\rfloor$$

In this equation, it can observed that we have to find the maximum value of intensity variation and add the value of $E_{\perp}/8$, where $E_{\perp}$ is the intensity variation in the direction that is perpendicular to the direction of the maximum intensity variation.

To find the maximum value of intensity variation, the function *compare* must be used. This function receives the values of the intensity variation of each direction that were computed in the previous part and returns the maximum absolute value, the absolute value of $E_{\perp}$ and a binary code indicating the direction of the edge of the figure. The codes for each edge direction are shown in table 1.

DIRECTION	CODE
Positive Horizontal	000
Negative Horizontal	001
Positive Vertical	010
Negative Vertical	011
Positive Right Diagonal	100
Negative Right Diagonal	101
Positive Left Diagonal	110
Negative Left Diagonal	111

Table 1 Codes for edge directions

When the outputs of the function *compare* are obtained, the magnitude described in the previous equation has to be found. The function used to obtain this value is the function *magnitude*, which is described below.

```
------------------------ Magnitude function -----------------------
function magnitude
    (A,B:  filter_out)
    return filter_out is
begin
   return (A + (B/8));
end magnitude;
```

Finally, to decide which pixel is part of an edge or which is not, the magnitude value has to be compared with a signal of the edge detector code called *threshold*. This comparison is made in the edge detector code. If the magnitude is greater than the threshold, the value 255 is assigned to the correspondent pixel establishing that it is an edge. Otherwise it is assigned a 0. This process is developed for all the pixels of the image.

```
if Mag >= threshold then
   temp := foreground;
else
   temp := background;
end if;
```

It can be observed in this part of the code that the value of the constants foreground and background are assigned to the variable *temp* after the comparison. These constants are initialized in the package of image processing.

```
--------declare constants
constant foreground:integer:=255;
constant background:integer:=0;
```

All the steps that are needed for doing the operations that require the Sobel edge detection algorithm have been explained in the previous part. However, this procedure is done for just one 3x3 window of the image. For that reason, some code is required to read the entire image in 3x3 windows and store the data obtained as a result of applying the Sobel operators to each of these windows. The code that reads the image and splits it in several 3x3 windows is described in the edge detector code as follows.

```
if busy1 ='1' then
   memory(x1,y1):=input;
   if count_r1=2 and y1=1 then
      busy2:='1';
   end if;
   y1:=y1+1;
   if y1=num_cols+1 then
      y1:=1;
      x1:=x1+1;
      count_r1:=count_r1+1;
   end if;
   if x1=4 then
      x1:=1;
   end if;
   if count_r1=num_rows then
      busy1:='0';
   end if;
end if;
```

In this part when *busy='1'* the image is read pixel by pixel and stored in the variable *memory*. Note that this variable is a 2-d array of 3 rows by *n* columns since the data is read row by row. We can observe that it is not necessary for the array to have a length equal to the size of the rows of the image because the processing is performed in a 3x3 window. When *busy='2'*, the data needed for convolving it with the masks has already been read, so the data stored in the variable *memory* is assigned to the variable *A*. Since the variable *A* is of size 3x3, it is necessary to apply the *mod 3* operation to the variable *count_r2* to preserve the order of the data of the image.

```
if busy2='1' then
   for j in 1 to 2 loop
      for i in 1 to 3 loop
         A(i,j):= A(i,j+1);
      end loop;
   end loop;
   for i in 1 to 3 loop

      x2:=mod3(count_r2+i-1);
      A(i,3):=memory(x2+1,y2);

   end loop;

   y2:=y2+1;

   if y2=num_cols+1 then
      count_r2:=count_r2+1;
      y2:=1;
   end if;
   if count_r2 = num_rows-2 then
      busy2 := '0';
   end if;
```

Note that the function *mod3* is used to apply the *mod 3* operation to *count*_r2. This function has been created because it is not implemented in the ISE 8.2 and can be found in the package *processing* with name *mod3*. Its implementation is shown below.

```
----------------------funcion mod3-------------------------------
function mod3( c: in integer)return integer is
variable a,b: integer ;
begin
   a:=c;
while a>=3 loop
   a:=a-3;
end loop;
   b:=a;
return b;
end function mode3;
```

IMPLEMENTING, COMPILING AND SIMULATING THE SOBEL EDGE DETECTION CODE

In this part, a description of all the required steps for implementing the Sobel Edge detection algorithm in VHDL code explained in this tutorial are provided.

Create A New Project

To do this go to the Project navigator, select File, new Project. In the window that appears right after doing so, write *sobel_edge_detection* as the name of the Project. You can change the Project location to the directory that you want to work on. In this case, the *Projects* directory in the root is used.

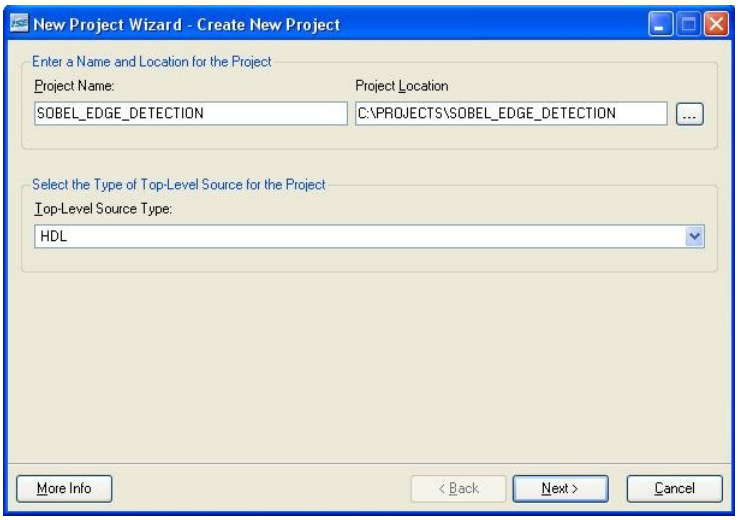

Figure 2 New project options.

Setting A Device

In the next window, you have to choose the type of device that you will be using. Since you are only going to simulate the design, you can just choose the specifications that appear in the next figure.

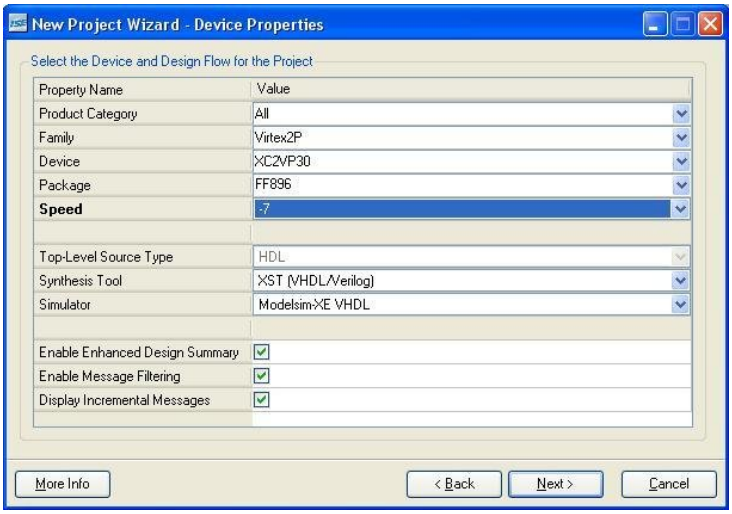

Figure 3 Device properties.

In the next window of the wizard of the new Project, since all the codes are provided, it is not necessary to create a new source. In the next window, add existing sources, you have to add the three codes that are provided to you. It is recommended that these codes are stored in the folder that you want to work on, in this case in the folder Projects. Then click on *next*.

Figure 4 Add existing sources.

The last window is the Project summary; you can verify all your specifications about your project. After clicking finish, a new window pops up. In this window you need to associate the codes with their respective purposes as shown in figure 5. Then, click *ok*.

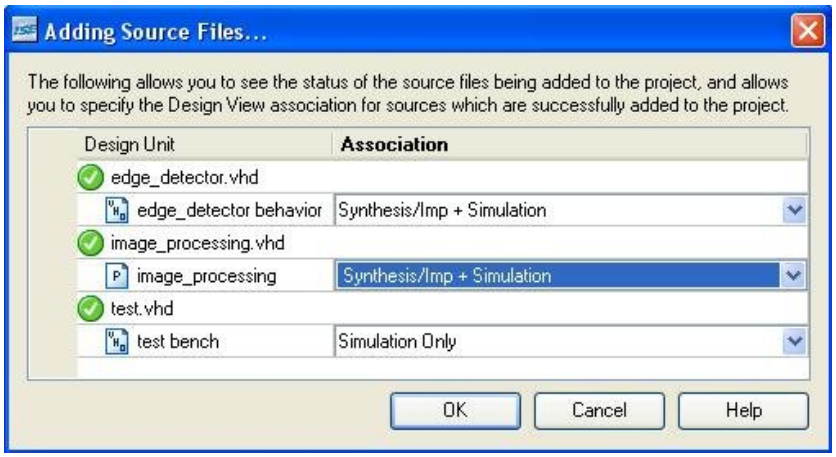

Figure 5 Status of the sources.

Note that all your codes appear in the library work as shown in the next figure. You can check the codes by double clicking on the name as shown below.

Figure 6 Sources in the libraries.

It is important that the image processing package, which contains all the functions of the algorithm, is saved in the *work* folder. The reason for that is that in the code of the edge detector the route of the image processing package is specified.

```
library IEEE;
use ieee.std_logic_1164.all;
use work.image_processing.all;
```

Synthesis

To compile the code you have to follow these steps: First, choose in *sources* *Synthesis*/*Implementation*. After that, select in the process window *Synthesize XST*. Double click on *check syntax*. When the compilation is successful, the icon of the Synthesize XST must be a check as shown below.

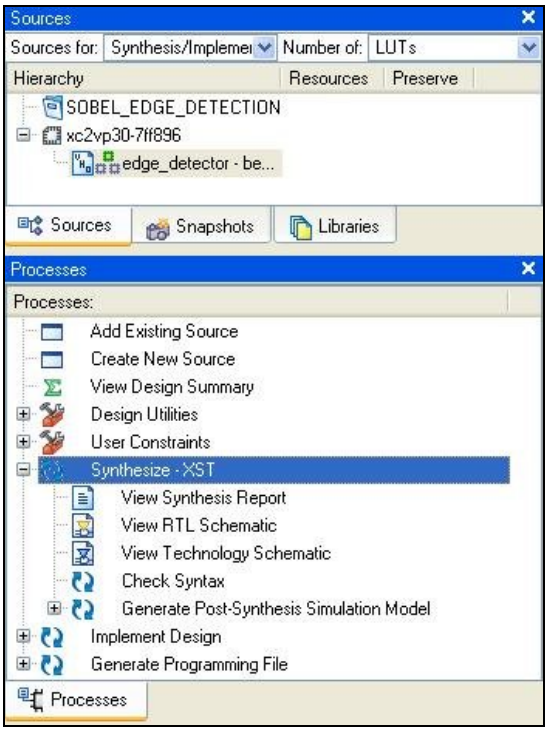

Figure 7 Running the synthesis process.

Simulation

When the compilation is successful you can simulate the project. In order to do this we use the *MODELSIM* tool. Certain parameters are set in the code of the test bench; these parameters can be modified by the user if it is necessary and are shown below.

Note: Remember that you can obtain the test bench code by double clicking on the name in *libraries* as shown before in the tutorial or you can select in *sources* the option *behavioral simulation* and double click in the name of the test bench that appears in the hierarchy.

The following parameters are specified. The size of the image is specified in number of rows and columns in the generic map of the entity of the test bench and so are the *hi_time* and *low_time* values of the clock.

```
entity test is
generic( HI_TIME:time:=75 ns;     -- high time in the clock
         LO_TIME:time:=25 ns;     -- low time in the clock
         num_rows :natural:=30;   -- number of rows in the input image
         num_cols :natural:=30;   -- number of columns in the input image
         wait_cycles :natural:=5 -- time required before outputs are
);                                -- written into frame buffers.
end test;
```

In this code, the sequence of values of the ports of some inputs is specified as shown below. Note that these values could be changed by the user if necessary.

```
Start<='0' after 0 ns,'1' after 5 ns,'0' after 105 ns;
threshold<=110 after 0 ns;
run<= transport '1' after 0 ns, '0' after 100000000 ns;
```

The names of the files that are used by the test bench are specified in this part of the text. Notice that you can change the filenames, but remember that the input txt file must be saved in the same Project folder where the two generated files are saved.

```
file imagein   :TEXT is in "imin.txt";     --input image file
file imageout  :TEXT is out "imout.txt";   --output file for storing magnitude outputs
file dirout    :TEXT is out "imdir.txt";   --output file which stores direction outputs
```

After modifying all the parameters in the test bench, you can simulate the code with a double click in *simulate behavioral model* as shown in figure 8.

Finally, you can check the result of the new image with the edges detected in the folder of the project with the name *imout.txt*, and the directions found in the process in the file named *imdir.txt.*

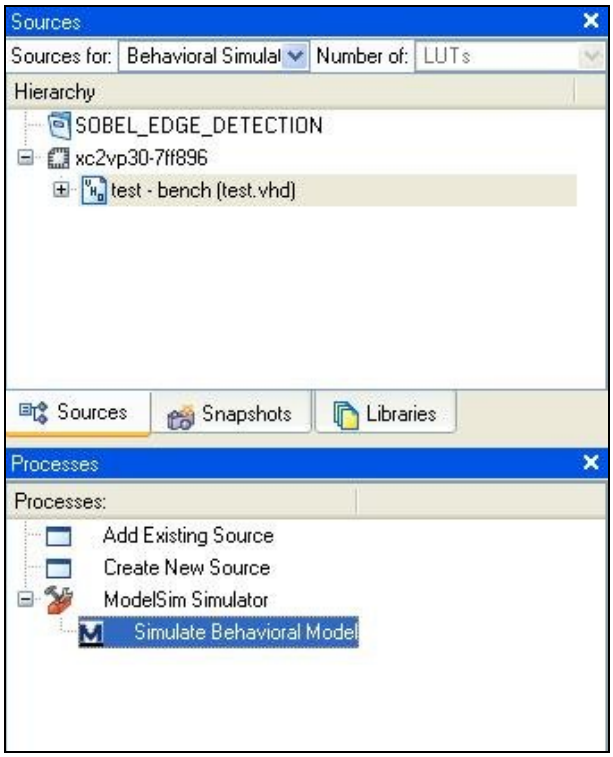

Figure 8 Running the simulation process.