Documentation of the Final Project of Mato and Massini Logic Networks

Project realized by:

Kevin Mato, Serial number: 845726 Person code 10499864 Luca Massini,

Serial number: 844049 Person code: 10504929

This project is a function that calculates the area in pixels of a portion of an image contained within an external RAM memory. The project was carried out as per specification using the VHDL language and verifying the possibility of synthesis, as per specification, on the Xilinx xc7a200tfbg484-1 board.

Project Choices

Algorithm

The algorithm used written in Matlab language.

```
function [result] = algo (arg)
        nr = 0 ;% default value
        nc = 0 ;% default value for number of columns
        s = 0; %threshold

        pos = 2; % reset counter address
        col = 0; % reset column counter
        line = 0; % reset line counter
        side1 = 0; % ram default values
        side2 = 0;
        line1 = 0;
        row2 = 0;

        nc = arg (pos); % assignments requesting memory addresses
        pos = pos + 1 ;% address counter increment
        nr = arg (pos);
        pos = pos + 1;
        s = arg (pos);
        looper = (nr * nc) + pos; % calculation of the number of cycles

        prime = 1; % flag to indicate if a pixel above threshold has not already been found, in vhdl it has been indicated as isFirst

        pos = pos + 1;

        while (pos <= looper)
                        if (arg (pos)> = s)
                                if (first == 1)
                                        side1 = col;
                                        side2 = col;
                                        row1 = row;
                                        row2 = row;
                                        prime = 0;

                                elseif (first == 0)
                                        if (with <side1)
                                                side1 = col;
                                        elseif (col> side2)
                                                side2 = col;
                                        end

                                        if (line> line2)
                                                row2 = row;
                                        end
                                end
```

```
                        end
                        col = col + 1; % increases the counters of the column
                        pos = pos + 1;
                        if (col> nc) % checks that the column is not outOfBound
                                col = 1;
                                row = row + 1;
                        end
          end
        if (first == 1)
                result = 0;
        else
                el = side2-side1 + 1;
                el2 = row2-row1 + 1;
                result = el * el2;
        end
else
        result = 0
end
```

The implementation started with the definition of a search algorithm, which in every possible solution has linear complexity in the worst case, due to the very nature of the problem. The algorithm used for our version of the project was not considered fundamental for the didactic purpose that in our opinion the project was intended, so we opted for a linear memory scan aiming at the use of the components learned in class. The algorithm scrolls the positions within the memory and checks if the pixel at that position is above the threshold, if so it compares its coordinates in rows and columns and saves the value if it determines a shift of the perimeter of the figure concerned, based on that moves the lowest or highest line or the right or left side of the rectangle surrounding the pixel area.

Architecture

First of all we thought about the realization of the project through a finite state machine and to divide the project into two sections, one called "Datapath" where the components are controlled by the logic of the state machine and where the real image data is collected and processed. The second is the FSM itself, which communicates with the Datapath through a signal that is the result of a logic component of the ALU and modifies its states according to a path on the graph determined by this last signal, and gives orders to the datapath through control signals.

Let's analyze the Datapath in detail: this section deals with implementing the more “passive” elements of the algorithm, ie that have only the task of saving data and keeping track of the position within the image. So in the algorithm we have highlighted the use of three counter variables. Made with counters not only with clock and reset signal but also a clock enable signal so that they could only be activated at certain times.

The position of a pixel is determined through three counters, one for the column ("col" is counter col), one for the row ("row" is counter row) and one for the memory address of that pixel (counter pos) . The memory address counter has a second reset "rst2" which brings pos to the value 0 since rst which is connected to all datapath resets would bring it to two while on col and line to 0. The enables are directly connected to the FSM since every “active” task is delegated to it.

For the storage of information we have placed a RAM with two inputs and two outputs, which is writable only on the first of the inputs. The RAM also has two control signals according to which if '1' the input is enabled, otherwise the corresponding output shows the default value zero.

The RAM stack is made up of 10 cells, the first two with default values with the neutral values of the operations, that is 0 and 1, useful for making unitary sums and reporting the null value in particular cases by the algorithm. In the other cells all the variables of the algorithm are saved, such as index of the first side, second side, upper row and lower row of the image.

The RAM output values are saved in 18-bit memory registers, just as the whole architecture has only 18-bit registers. The choice of 18 bits was a choice of implementation convenience, also due to the possibility of code reuse. The registers are of the parallel-parallel type since saving the value requires only one clock cycle.

The registers were placed for three reasons:

1. Trying to virtually reduce the critical paths in the datapath, thus putting save areas that would have allowed us to break the switching times of the circuit going up to the ALU, over multiple clock cycles.
2. If you can break the switching over multiple cycles, then we thought you could decrease the clock periods and thus increase the frequency.
3. The registers were also the basis for an eventual optimization of the state machine, since by slightly modifying the architecture and the control signals it could be made similar to a pipeline architecture.

Attached to the registers is the ALU. It is the bottleneck of the entire project. Its multilevel structure is the result of a discussion aimed at saving the area, given that the base frequency for the project was 15 ns and our project has no problems up to about 9.5 ns, we aimed at saving d 'area. It must also be said that the project does not aim to be excellent in saving time or even in the area, but we wanted to build the most scalable architecture ever that the time available and the experience would allow us, with an eye on small changes in the future.

At the output of the ALU we find a sector for recording the results, where a register for the results with a numerical meaning and a flip-flop for the results of a logical / Boolean type derived from comparison operations are respectively placed.

The former has only one feedback link on the RAM while the Boolean ends up directing the FSM.

The scrolling of data between a component is not left to chance, demultiplexers and multiplexers are involved in the datapath always controlled by the FSM to change the data paths based on which they must exit the entity, be manipulated through operations or saved in the RAM memory .

Demultiplexers are usually indicated with a letter D and a number for their enumeration. They have one input and are controlled by a logic signal that indicates which of the outputs to bring the input to. The multiplexers are called super and numbered in turn, they have 4 inputs called "super" and are controlled by a vector of signals for the selection of the outgoing one. The latter two carry data in 8-bit vectors, while there is a last multiplexer that has 8-bit inputs called mux8BIT and is the component that determines whether the most significant or least significant bits are written when the result is written.

Advantages and disadvantages

Our idea for the project has always been to create something in the most professional way we could, with the most correct and rigorous methods we knew, it is no coincidence that we have deepened the discourse of this project with several readings, and that would allow us to learn some reusable thing in the world of work. Our project therefore proposes itself as an absolute objective scalability. By Scalability we mean that, given that our architecture is based on a datapath and an FSM, all of this is highly scalable. In fact, if you want to change the algorithm, it would be enough to modify only the FSM that gives the control signals to the datapath and if you want to start any new project, the datapath could be an excellent starting point to connect a new FSM that implements the algorithm of the proposed project . Our design may not be the fastest given the choice of an algorithm that has worst case performance in the best case but we didn't care. Having chosen a bad algorithm, we opted for something that, taking into account the scalability of the architecture, could be the most contained idea in the area that we could. We have refused the use of processes as long as possible as we knew they tend to give very broad results in the area. We tried to write an fsm that could give us freedom of optimization by setting the various command signals ourselves, one by one. Overall we are satisfied with what we have been able to do, above all because we also have temporal performances in the submodules tested separately that make us proud, a little less on the ALU submodule that we have designed for our computations and that we have built on 8 levels trying to save area.

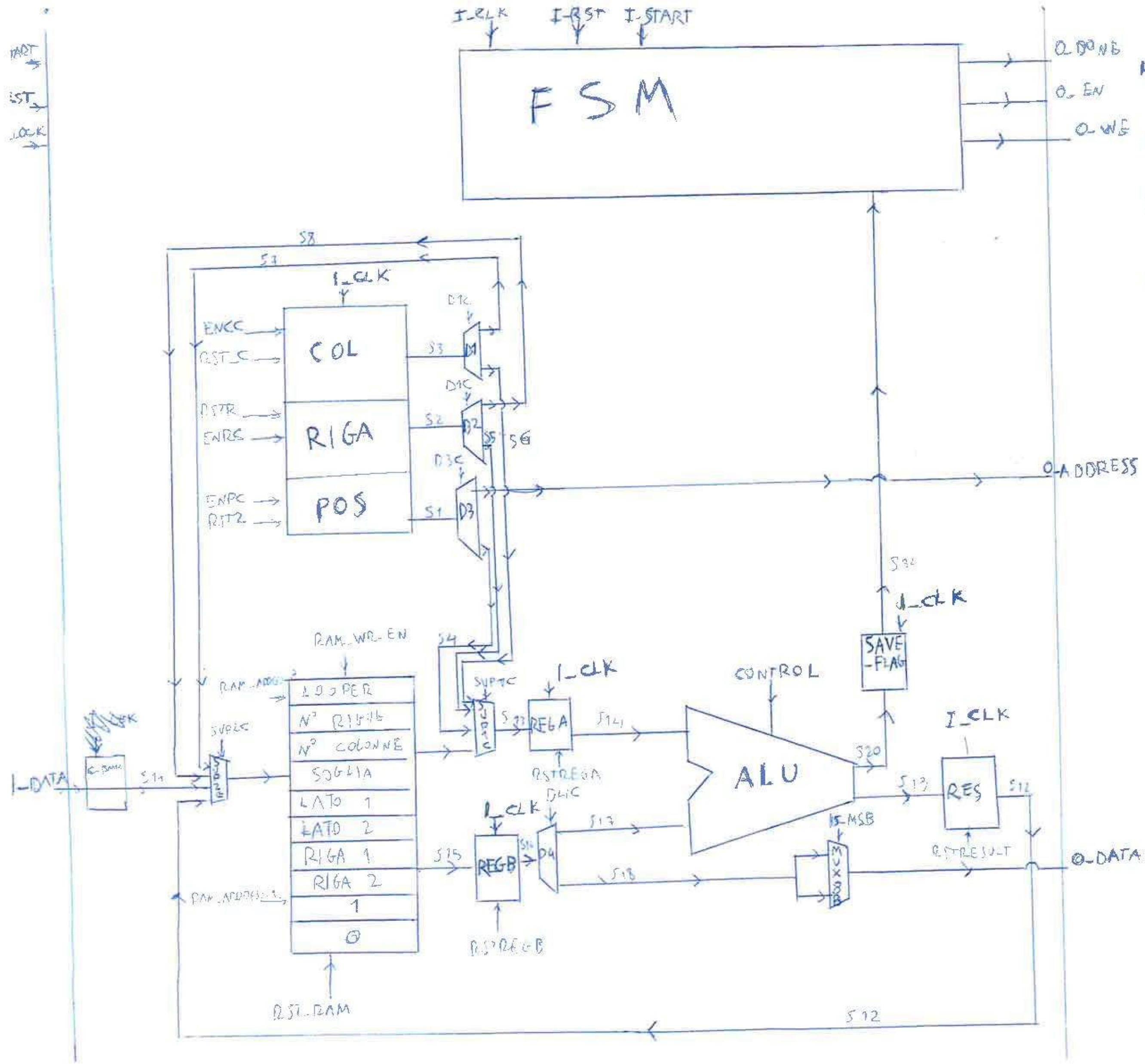
Design summary

The conceptual design was carried out following a top down methodology in which we found how many portions to divide the architecture recursively and so on up to the individual components. For each version of the project we designed to better imagine the result and once we were sure of what we needed, we created each single component separately, we tested it and then combined it into sub-modules according to the usefulness they had, and then also test they separately, the final stage involved joining them together and joining them with an FSM

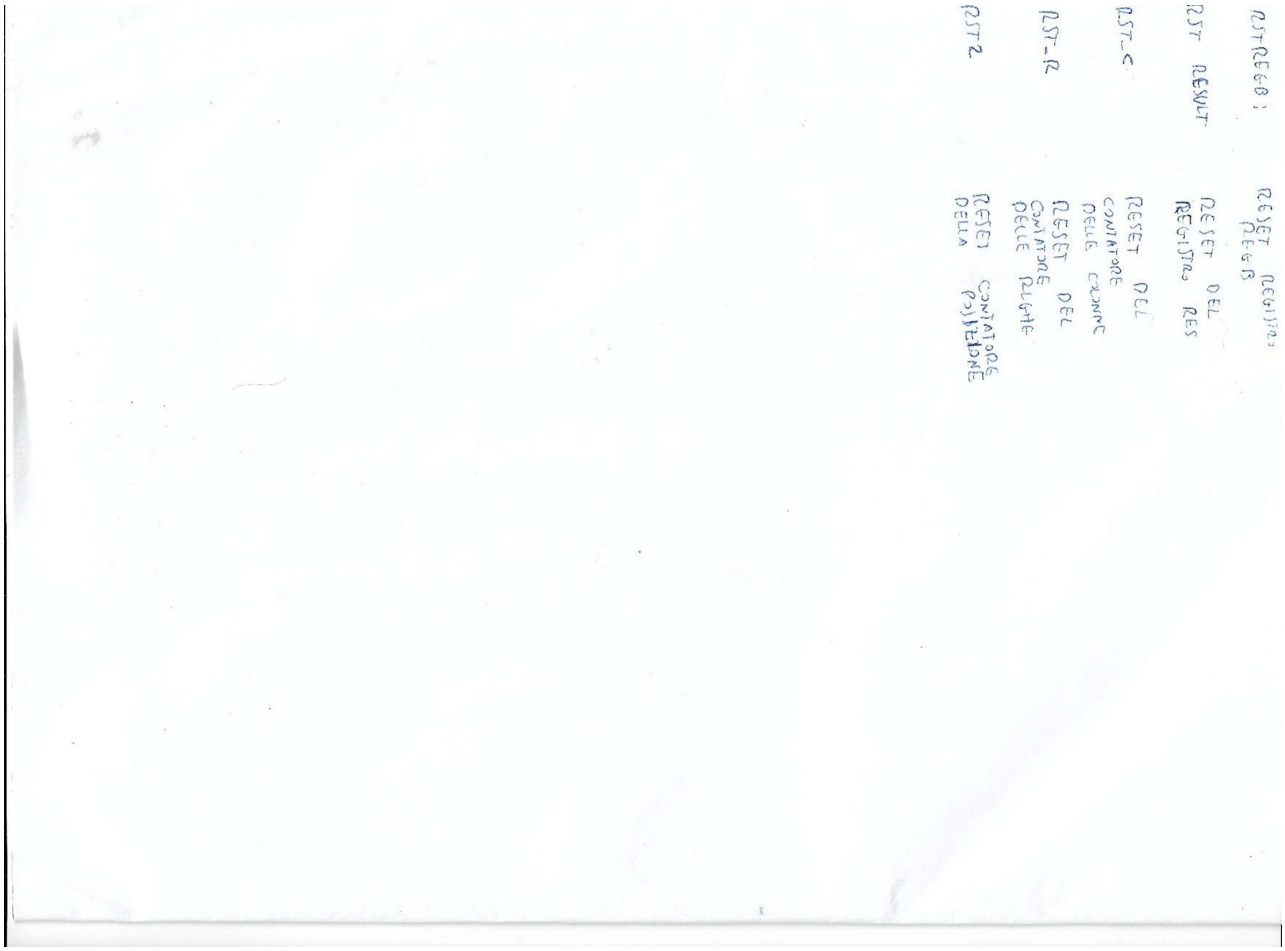
initially not very ripe. Once the control signals of the FSM have been aligned with its datapath and we have verified the correctness of the design.

The difficulties encountered in the project were mainly in the use of the tools and in the search for the best way to define a new project for us in a language like VHDL.

Representation



LEGENDA FSM	SEGNALI DI OUTPUT DELLA FSM
RAM_ADDRESS_0	INDIRIZZO DELLA MEMORIA RAM DEL PROGETTO RELATIVO ALLA PRIMA PORTA
RAM_ADDRESS_1	INDIRIZZO DELLA MEMORIA RAM DEL PROGETTO RELATIVO ALLA SECONDA PORTA
RAM_WR_EN	SEGNALE DA DARE ALLA RAM PER POTER SCRIVERE IN ESSA
SUP1C	SEGNALE DI CONTROLLO PER IL MULTIPLEXER SUPER1
SUP2C	SEGNALE DI CONTROLLO DEL MULTIPLEXER SUPER 2
D1C	SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D1
D2C	SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D2
D3C	SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D3
D4C	SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D4
IS_MSB	CONTROLLO DEL DEMUX MUX8B.
ALU_CONTROL	CONTROLLO DELLA ALU
ENRC	SEGNALE DI ENABLE PER IL CONTATORE DELLE RIGHE
ENCC	SEGNALE DI ENABLE PER IL CONTATORE DELLE COLONNE
ENPC	SEGNALE DI ENABLE PER IL CONTATORE DELLA POSIZIONE
EN_PORT_0	ENABLE PER LA LETTURA DELLA RAM DALLA PRIMA PORTA
RST_RAM	SEGNALE PER IL RESET DELLA RAM AI VALORI DI DEFAULT
RST REGA	RESET DEL REGISTRO REGA



TEST

Here are the tests performed on the VHDL component for its validation:

NB: the tests presented here are in post synthesis.

1.

This is the fourth test given to us as an example test: Values given

to the testbench ram:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00010101", 30 => "00000011",
31 => "00000011", 32 => "00000011", 33 => "00000011", 36 => "00000111", 37 => "00000111",
38 => "00000111", 39 => "00000111", 42 => "00001011", 43 => "00001011", 44 => "00001011", 45 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111", 72 => "00011001", 78 => "00000011",
79 => "00000011", 80 => "00000011", 84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011", 108 => "00000111", 114 => "00011111",
120 => "00011001", 126 => "00000011", 132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011", 144 => "00001111", 145 => "00001111",
146 => "00001111", 147 => "00001111", others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00010101" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

2.

This is the third test given to us as an example test:

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00000010",
30 => "00000011", 31 => "00000011", 32 => "00000011", 33 => "00000011",
36 => "00000111", 37 => "00000111", 38 => "00000111", 39 => "00000111",
42 => "00001011", 43 => "00001011", 44 => "00001011", 45 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111",
72 => "00011001", 78 => "00000011", 79 => "00000011", 80 => "00000011",
84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011",
108 => "00000111", 114 => "00011111", 120 => "00011001", 126 => "00000011",
132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011",
144 => "00001111", 145 => "00001111", 146 => "00001111", 147 => "00001111",
others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "01101110" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

3.

This is the second test given to us as an example test: Values given

to the testbench ram:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00000000",
30 => "00000011", 31 => "00000011", 32 => "00000011", 33 => "00000011",
36 => "00000111",  array (65535 downto 0) of std_logic_vector(7 downto 0)  00000111",
42 => "00001011",                                                          00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111",
72 => "00011001", 78 => "00000011", 79 => "00000011", 80 => "00000011",
84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011",
108 => "00000111", 114 => "00011111", 120 => "00011001", 126 => "00000011",
132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011",
144 => "00001111", 145 => "00001111", 146 => "00001111", 147 => "00001111",
others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "10101000" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

4.

This is the first test given to us as an example test: Values given to

the testbench ram:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "11111111",
30 => "00000011", 31 => "00000011", 32 => "00000011", 33 => "00000011",
36 => "00000111", 37 => "00000111", 38 => "00000111", 39 => "00000111",
42 => "00001011", 43 => "00001011", 44 => "00001011", 45 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111",
72 => "00011001", 78 => "00000011", 79 => "00000011", 80 => "00000011",
84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011",
108 => "00000111", 114 => "00011111", 120 => "00011001", 126 => "00000011",
132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011",
144 => "00001111", 145 => "00001111", 146 => "00001111", 147 => "00001111",
others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

5.

The following test is for the case where all image values are above the threshold. In this case, the threshold is set to zero and the number of rows and the number of columns are set to 4 and 3 respectively. In the ram, each memory cell is assigned to the conventional value above the threshold value, that is 214 (11010110 in binary). The result obtained is in fact equal to 12 or the product of the number of rows and the number of columns.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00000011", 3 =>"00000100",4 => "00000000",
others => ("11010110"));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00001100" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

6.

The following test relates to the case in which the number of columns of the portion of the image to be considered is equal to zero.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00000000", 3 =>"01001100",4 => "00001100",
others => ("11010110"));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

7.

This test is for the case where the number of rows is zero. Obviously it is consequently expected, as positively found in the result of the test, that the minimum area of the figure of interest is also zero.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00001110", 3 =>"00000000",4 => "00001100",
others => ("11010110"));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

8.

This is the test where both number of columns, rows, threshold and values stored in the ram are zero. The area of the figure of interest is also expected to be zero as a result, as positively found in the testbench result.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00000000", 3 =>"00000000",4 => "00000000",
others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

9.

This test represents the case where all values are above threshold. The number of rows and columns is equal to 31 in both cases, the threshold is equal to 4. The result is therefore expected to be the product of the number of rows and the number of columns or 961 (ie 31 x 31). The difference, not just, between this case, and the analogous test in which you always had all the pixels above the threshold, is that in this case you also test the eight most significant bits of the ram (1) and only the 8 least significant bits ram (0) as was done in the other tests including those given to us as an example test.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00011111", 3 =>"00011111",4 => "00000100",
others=>("00000111"));
```

Result obtained:

```
assert RAM(1) = "00000011" report "FAIL high bits" severity failure;
assert RAM(0) = "11000001" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

10.

The following test relates to the case in which there is only one pixel above the threshold and in which the number of rows and columns is the maximum one, i.e. 11111111 in binary, i.e. 255 in decimal. It is expected, as subsequently positively found in the test, that the value of the area of the figure of interest is 1 in ram (0).

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "11111111", 3 =>"11111111",4 => "00000100",
7=>conv std logic vector(18,8), others=>(others=>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000001" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

11.

The following test relates to a general ram input in which, however, inside the figure (number of rows = 4 and number of columns = 5 and threshold equal to 4) the pixels above the threshold form a rectangle trapezoid positioned in such a way that the height (of value equal to 3) is horizontal and that the greater base (of value equal to 4) is to the left of the smaller one. The minimum area of the rectangle that circumscribes this figure in this case is equal to 12, equal to the product of the greater base by the height of the trapezoid, as positively found by the test result.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00000101", 3 =>"00000100",4 => "00000100",
7=>conv_std_logic_vector(5,8), 8=>conv_std_logic_vector(9,8),
9=>conv_std_logic_vector(5,8), 12=>conv_std_logic_vector(12,8),
13=>conv_std_logic_vector(17,8),14=>conv_std_logic_vector(5,8),
17=>conv_std_logic_vector(8,8),22=>conv_std_logic_vector(5,8),
18=>conv std logic vector(8,8), others=>(others=>'0'));
```

Result obtained

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00001100" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

12.

The following test relates to the case in which all the values of the figure of interest are equal to the threshold. The number of rows and columns and the threshold is set equal to 15. The value of all the elements within this figure are equal to the threshold

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00001111", 3 =>"00001111",4 => "00001111",
others => ("00001111"));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "11100001" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

13.

In this test we are in the case in which the number of columns is equal to 5 and the number of rows is equal to 4 and the threshold is set equal to 3. The bits above the threshold inside the figure are placed as a right triangle with major side equal to 5 and minor side equal to 4. The minimum rectangle that circumscribes the area has an area equal to 20 as then confirmed by the test.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00000101", 3 =>"00000100",4 => "00000011",
5 => conv_std_logic_vector(4,8), 9 => conv_std_logic_vector(5,8),
24 => conv_std_logic_vector(6,8),others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00010100" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

14.

In the following test we are in the case in which the only elements above the threshold are the vertices of the figure of interest. In particular, the first vertex is at the top left and the second at the bottom right. The number of rows and the number of columns are set respectively equal to 4 and 5. The threshold is set equal to 3. The value of the minimum rectangle area that is expected is 20 as then confirmed in the test result.

Values given to the testbench ram:

```
signal RAM: ram_type := (2 => "00000101", 3 =>"00000100",4 => "00000011",
5 => conv_std_logic_vector(4,8),  24 => conv_std_logic_vector(6,8),
others => (others =>'0'));
```

Result obtained:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00010100" report "FAIL low bits" severity failure;


assert false report "Simulation Ended!, test passed" severity failure;
```

Limit frequency

```
create_clock -period 9.6 -name i_clk [get_ports i_clk]
```

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,039 ns	Worst Hold Slack (WHS):	0,264 ns	Worst Pulse Width Slack (WPWS):	4,300 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	869	Total Number of Endpoints:	869	Total Number of Endpoints:	316

All user specified timing constraints are met.