HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF ELECTRONIC AND TELECOMMUNICATION

LSI CONTEST

REGIONAL MAXIMA

Team name: EDABK - HUST

Team members:

Pham Quang Anh	K63	Hanoi University of Science and Technology
Nguyen Viet Thi	K63	Hanoi University of Science and Technology
Nguyen Duc Quang	K63	Hanoi University of Science and Technology

Hanoi, 01-2023

INFORMATION PAGE

1. Team name: EDABK – HUST
2. Task level of design: Level 3 (Advanced) - Improved algorithm, unlimited image size can be processed
3. Members' information:

- Pham Quang Anh
  - School's name: Hanoi University of Science and Technology
  - Grade: 63
  - Address: No. 1 Dai Co Viet street, Hai Ba Trung, Hanoi, Vietnam
  - Mail address: anh.pq182359@sis.hust.edu.vn
  - Size of T-shirt: M
- Nguyen Viet Thi
  - School's name: Hanoi University of Science and Technology
  - Grade: 63
  - Address: No. 1 Dai Co Viet street, Hai Ba Trung, Hanoi, Vietnam
  - Mail address: thi.nv182798@sis.hust.edu.vn
  - Size of T-shirt: XL
- Nguyen Duc Quang
  - School's name: Hanoi University of Science and Technology
  - Grade: 63
  - Address: No. 1 Dai Co Viet street, Hai Ba Trung, Hanoi, Vietnam
  - Mail address: quang.nd182736@sis.hust.edu.vn
  - Size of T-shirt: XL

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

CHAPTER 1. NEW PROPOSED ALGORITHM

This chapter will show the restrictions of Regional Maxima algorithm was mentioned in design specification and propose a new algorithm which can remove those restrictions but still give correct result.

1.1 Restrictions of original algorithm

With the original algorithm, the number of iterations needed to get the final result is unknown, because we don't know when **output** is equal to **output_cp**. In the case there are many elements in a region have the same value, checking when **output** is equal to **output_cp** will increase the number of iterations significantly.

```
input_image =

     5     5     5     5     5     5
     5     5     5     5     5     5
     5     5     5     5     5     5
     5     5     5     5     5     5
     5     5     5     5     5     5
     5     5     5     5     5     6

>> output = lsi_imregion_max(input_image)
Loop 1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     0
     1     1     1     1     0     0
     1     1     1     0     0     1

Loop 2
     1     1     1     1     1     1
     1     1     1     1     1     0
     1     1     1     1     0     0
     1     1     1     0     0     0
     0     1     0     0     0     0
     1     0     0     0     0     1

Loop 3
     1     1     0     0     0     0
     1     1     1     0     0     0
     0     1     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     1

Loop 4
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     1

Loop 5
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     1
```

Figure 1.1 Matlab result in case of large number of loops

Error! Reference source not found. is the result in Matlab of example of the case there are many elements in a region of matrix have the same value. As we can see, the algorithm need to run 5 loops until the end. Each loop contains 36 iterations over 36 elements of the matrix. In total, 180 iterations are required for this case. For hardware, if each element of matrix needs one cycle to traverse, the algorithm spend 180 cycles running to get the final result.

The best case is when all elements of the matrix have the same value, number of iterations in this case is equal to number of elements. Thus, except for the best case, the number of iterations (or cycles for hardware) is always greater than the number of elements of the matrix.

1.2 New proposed algorithm

In this section, we will propose an algorithm which can remove above restrictions. The biggest goal of modifying algorithm is to shorten execution time by reducing number of iterations. Or in hardware's parlance, this means reduce the number of cycles to get the final result. In the proofs below, we will show that this number of iterations is always fixed and equal to the number of image elements for all cases of different image's contents. Of course, this algorithm must ensure **output** matrix will be correct by reasoning and simulating at the software level.

1.2.1 Main idea

Detailed goal of our new algorithm is to replace or remove checking when **output** is equal to **output_cp**, because as analyzed above, this operation will cause number of iterations to increase.

In the simplest case when we consider an element of the matrix, basic principle of original algorithm is to find regional maxima in an area whose size is equal to window's size (3x3) as described in Figure 1.2. In this case, the task of the algorithm is to check if the element colored in red in the matrix is greater than all the elements in yellow.

0	0	0	1	2	3
0	1	0	2	3	3
0	0	0	2	3	4
1	3	1	5	4	5
0	0	0	3	2	4
2	1	0	1	2	3

Figure 1.2 The simplest case of finding regional maxima

But Figure 1.2 is just the simplest possible case. If we are looking at the element "2" at the position of (row 1, column 5), we will see that in the 3x3 area around this element, there is an element "2" at the position of (row 2, column 4). Continue, we will see around the element "2" in (row 2, column 4), there is an element "2" at the position of (row 3, column 4) is in its 3x3 area. At this time, the task of the algorithm is to check if the value of all red elements is greater than the yellow elements surrounding them, not just check for a single red element.

0	0	0	1	2	3
0	1	0	2	3	3
0	0	0	2	3	4
1	3	1	5	4	5
0	0	0	3	2	4
2	1	0	1	2	3

Figure 1.3 The case when the area to be considered is expanded

Figure 1.3 explains why the original algorithm needs to check the condition 2 "(the value of pixel A in any 8-neighborhood) = (the value of the center pixel)" and the stopping condition of the algorithm is "**output** == **output_cp**". This means, the purpose of these two conditions is to ensure that all equal neighboring elements in the input image will generate the same value for the corresponding elements in the **output** matrix. In the case of Figure 1.3, elements at positions (rows 1, column 5), (row 2, column 4), (row 3, column 4) of matrix **output** will have value 0 because the value of red elements in Figure 1.3 is not geater than their sorrounding yellow elements.

Applying the analysis above, instead of checking when the **output** is equal to **output_cp**, we will expand considered area as shown in Figure 1.3. The corresponding values on the output matrix of the considered elements (red elements as shown in the Figure 1.3) will be updated at the same time. This ensures that all red elements will have the same value without using condition 2 "(the value of pixel A in any 8-neighborhood) = (the value of the center pixel)" and stopping condition "**output** == **output_cp**".

A condition for this new algorithm is which elements have already been checked will not be checked again. For example, if the element in (row 3, column 4) of the output matrix has been updated, then when browsing to (row 3, column 3), we will not go to (row 3, column 4), but switch immediately to browse the position of (row 3, column 5).

The detailed operations of the algorithm will be presented in sections 1.2.2 and 1.2.3.

1.2.2 Flow chart

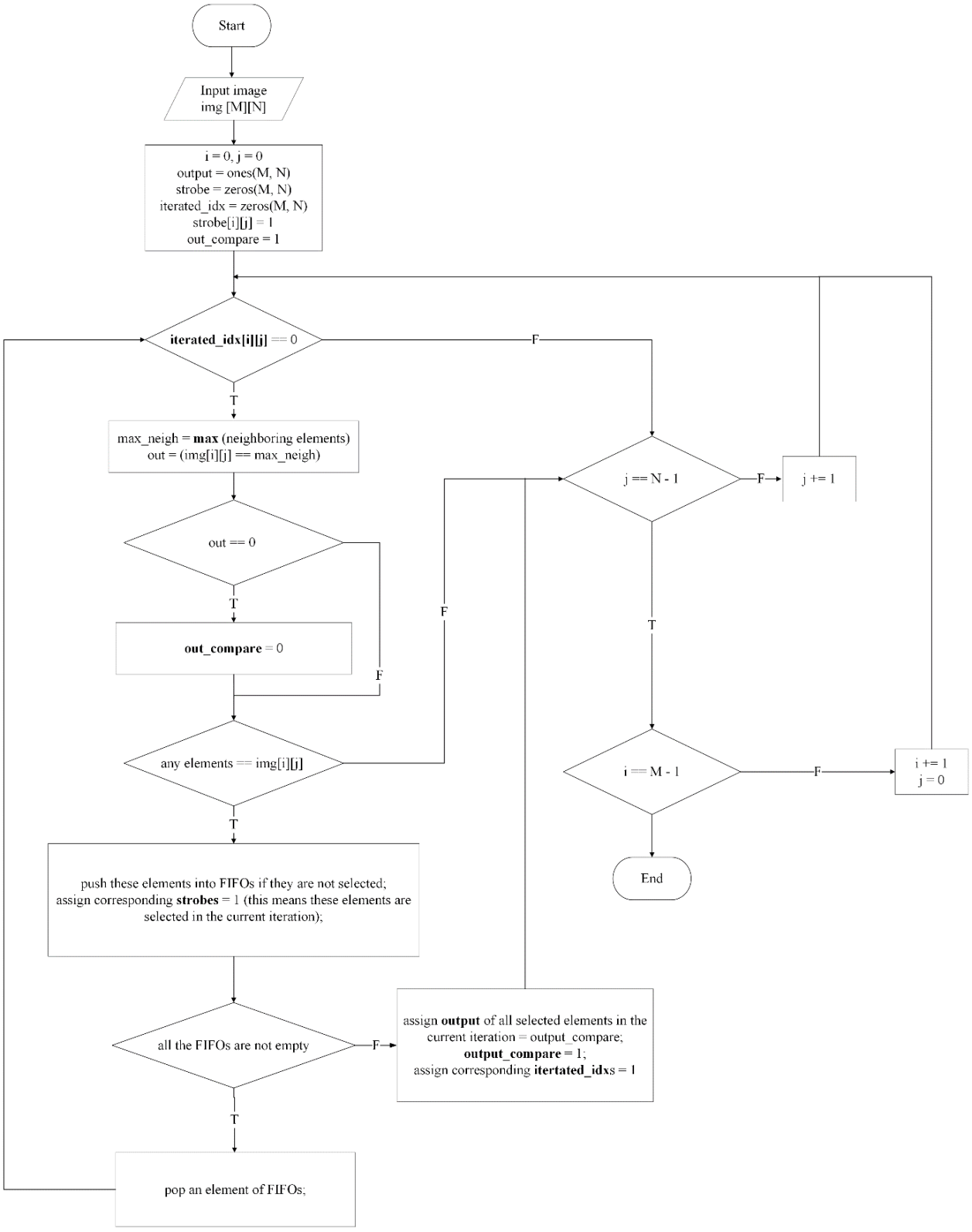

Figure 1.4 Proposed algorithm flow chart

Note:

- i , j: Index of element
- ones: assign all element of matrix to 1
- zeros: assign all element of matrix to 0
- iterated_idx: The matrix that mark the elements of input image are iterated or not, each element of iterated_idx matrix corresponds to an element of input image.
- out_compare: Temperal compare result of current iteration. After the end of current iteration, all selected elements are assigned out_compare.
- strobe: The matrix that mark the elements of input image are selected in current iteration or not, each element of strobe matrix corresponds to an element of input image.

1.2.3 Example

- **Process 1**: Prepare matrix **input** to store the input image, matrix **output** initialized with true (1), matrix **strobe** to mark the selected elements in the current iteration, initialized with false (0), matrix **iterated_idx** to mark the iterated elements, initialized with fale (0) as shown in Figure 1.5.

0	0	0	1	2	3
0	1	0	2	3	3
0	0	0	2	3	4
1	3	1	5	4	5
0	0	0	3	2	4
2	1	0	1	2	3

input[36]

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1

output[36]

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

strobe[36]

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

iterated_idx[36]

Figure 1.5 Process 1 of the new algorithm

- **Process 2**: Start with element at position (row 1, column 1). The red element of matrix **input** in Figure 1.6 is the selected element in the current iteration. The yellow elements of matrix **input** in Figure 1.6 are the neighbors of selected element. At the time when element (row 1, column 1) of **input** is selected, position (row 1, column 1) of

matrix **strobe** will be updated to true (1) as shown in Figure 1.6. Also, the element at position (row 1, column 1) of matrix **iterated_idx** will be updated to true (1) too. This means element at position (row 1, column 1) is iterated, and will never be traversed or re-selected in any future iterations.

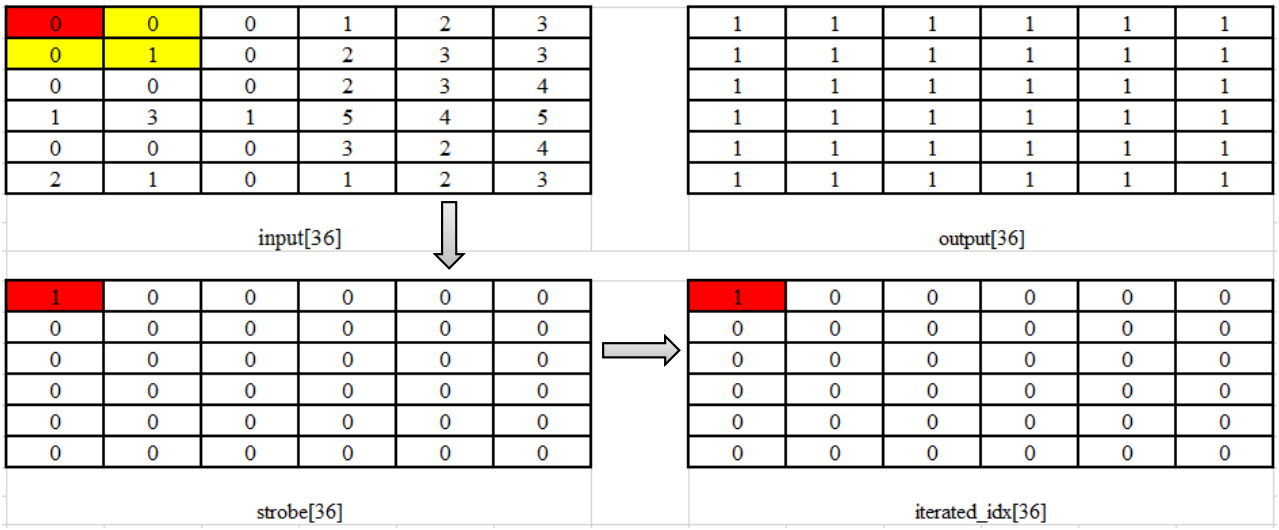

Figure 1.6 Iterate element in position row 1, column 1

The task now is to check if any of the neighbors are equal to the red element. We will see the elements at position (row 1, column 2) and (row 2, column 1) are equal to red element. So, the elements (row 1, column 2) and (row 2, column 1) now are the selected elements too. Neighboring elements now are all the elements colored yellow in matrix **input** as shown in Figure 1.7 because we add neighboring elements within 3x3 area of 2 new red elements. This process is called expanding area process.

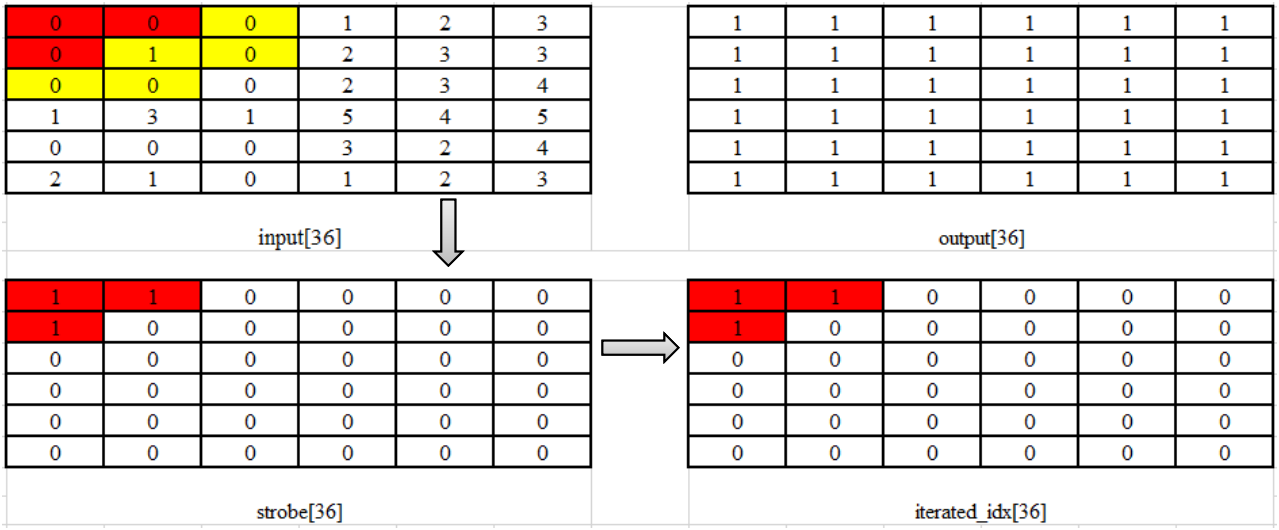

Figure 1.7 The first step of expanding selected area

Once again, we see some neighbors of elements (1, 2) and (2, 1) have value "0" too (positions (1, 3), (2, 3), (3, 1), (3, 2)). The selected area will continue to expand, the selected elements will continue to be added. And this process only stops when there are no more neighbors of the matrix **input**'s red elements have value "0". Matrices **input**, **strobe**, **iterated_idx** now are updated as shown in Figure 1.8.

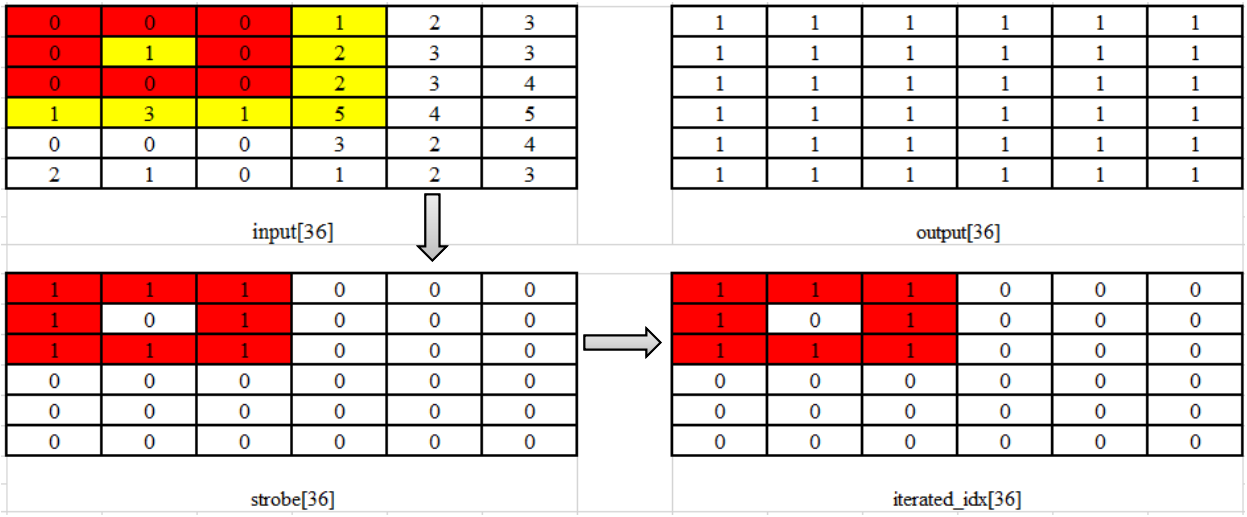

Figure 1.8 The last step of expanding selected area

In case no need to expand the selected area, just go to process 3.

- **Process 3**: After expanding the selected area is finished, the next task is to check if the red elements in matrix **input** have greater value than all the yellow elements. In this case, it's false. We will see in the matrix **strobe** which positions have a value of "1", then we will update value "0" for all elements with the corresponding positions on the matrix **output**. The value of matrix output now is shown as Figure 1.9.

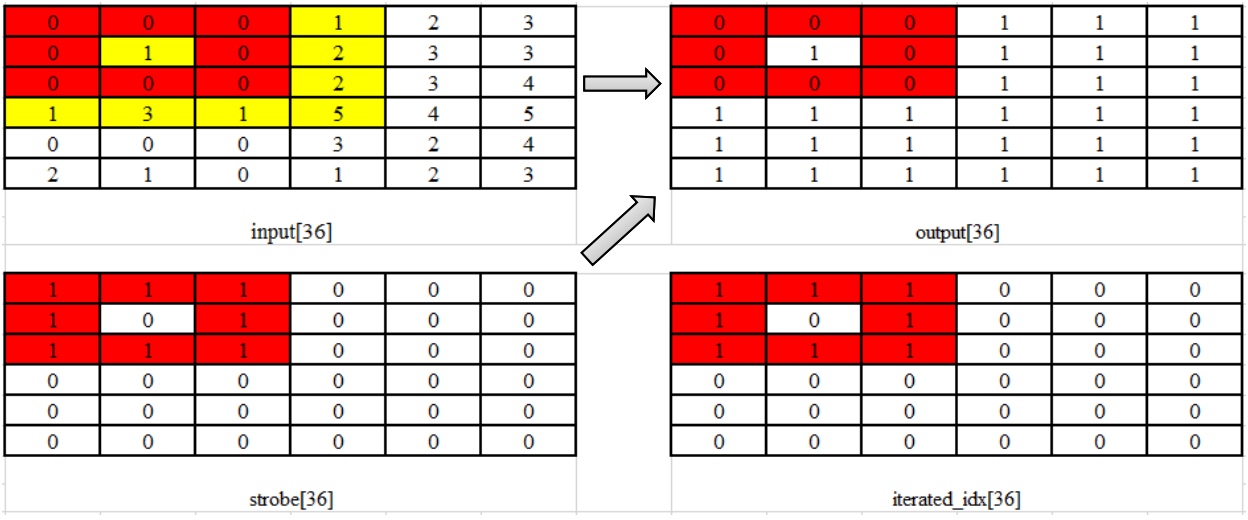

Figure 1.9 Process update output

- **Process 4**: The last iteration is finished. Matrix strobe now is cleared. Current status of four matrices are shown as Figure 1.10.

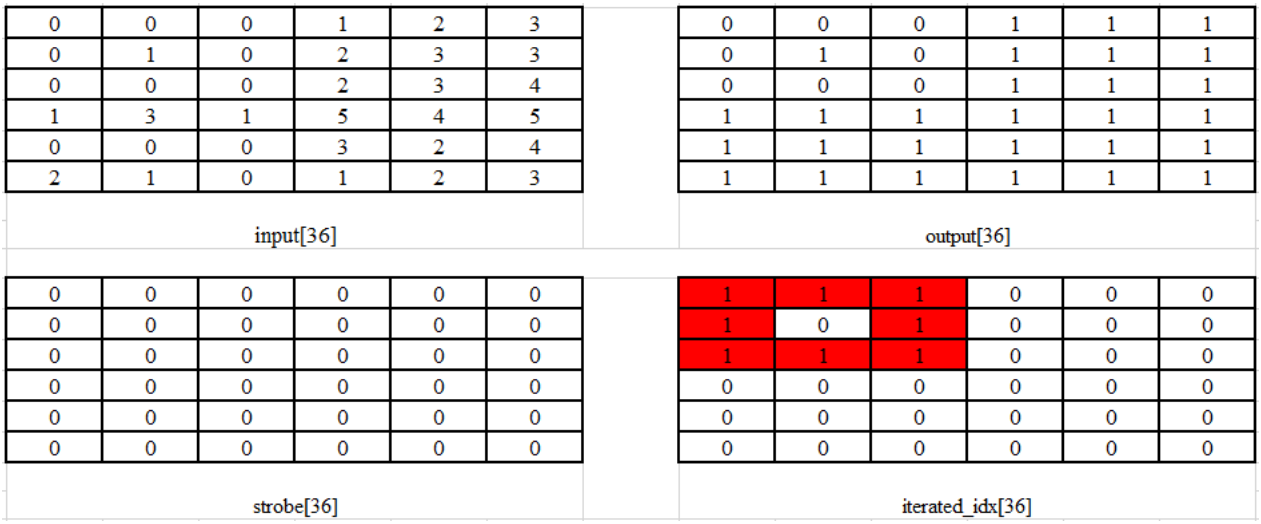

Figure 1.10 Status of four matrices after the first iteration

In this process, we need to check if all elements of matrix **iterated_idx** is "1". If not, the algorithm have not done yet, and we will go to the next iteration.

Note: By applying operation **selecting the next element to be considered** in 1.2.4, the algorithm will ignore elements whose corresponding values in matrix iterated_idx is "1". Example, in this case, elements (row 1, column 2), (row 1, column 3) have corresponding values in matrix iterated_idx is "1". So, we will skip them. The next iteration is to consider element (row 1, column 4).

From now on, process 2 and 3 will be repeated. So, after process 3, status of four matrices now are updated as shown in Figure 1.11.

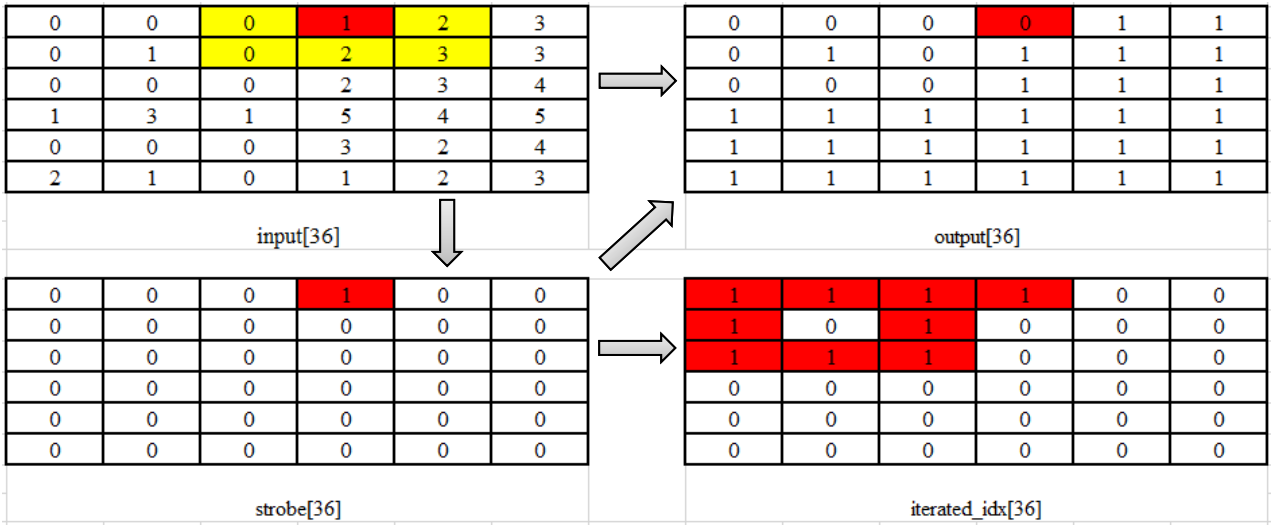

Figure 1.11 The next iteration

If all elements of matrix **iterated_idx** are “1”, the algorithm is finished. In other words, the stopping condition of the algorithm is that all elements of **iterated_idx** are equal to “1”. We can say that, after traversing all elements of matrix **input**, all elements of matrix **output** have been updated correctly. The result is shown as Figure 1.12.

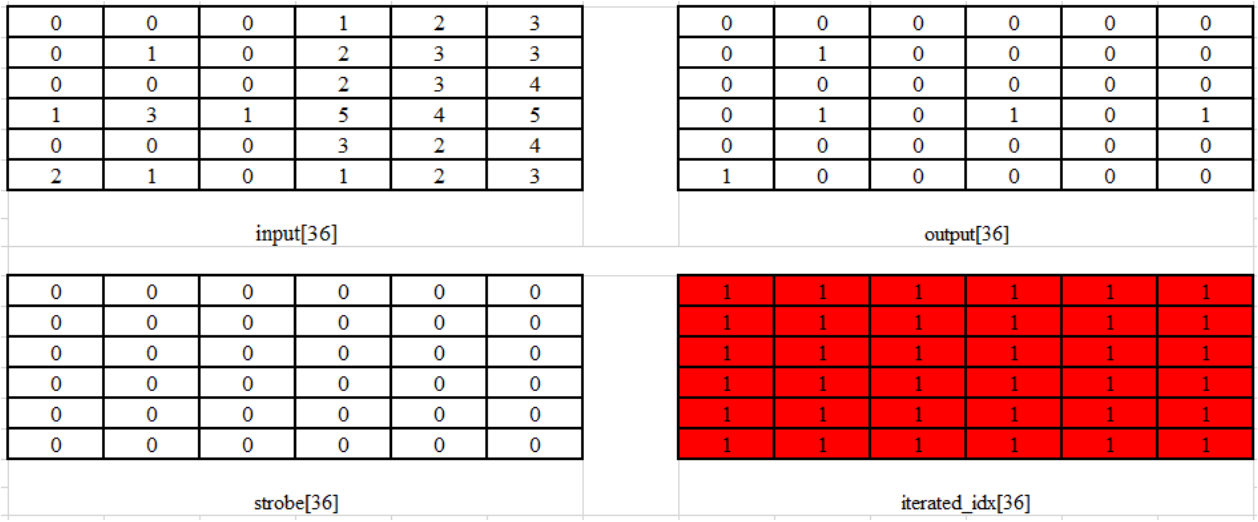

Figure 1.12 The result of the new algorithm

1.2.4 Operation selecting the next element to be considered

If we iterate through each element of matrix **input** in turn without skipping the elements which have the corresponding **iterated_idx** equal to “1”, redundant iterations will appear. For example, consider the example in 1.2.3, after the first iteration, elements (1, 1), (1, 2) are obviously already selected from the previous iteration, but if we keep doing in turn without skipping them, this will cause these 2 elements to be selected again redundantly. This is also a restriction of the original algorithm if you look at the Matlab code.

We propose an operation **selecting the next element to be considered** to solve this problem. The idea is like this:

- Through the matrix **iterated_idx**, we know which elements have been selected before. So our job is simply to find the row position of **iterated_idx** where at least one element is equal to "0" and on that row, find the position of the first zero element from left to right.
- The position of the row and column just found is the position of the next selected element.

For example, if we know the state of iterated_idx is like Figure 1.13, we will see the row position of **iterated_idx** where at least one element is equal to "0" is row 3 and the position of the first zero element from left to right on row 3 is column 4. So, the next selected element's position is (3, 4).

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

iterated_idx[36]

Figure 1.13 Example of operation selecting the next element to be considered

1.2.5 Impovements and trade-offs of new algorithm

1.2.5.1 Improvements

- As we can see in section 1.2.3, number of iterations is **always** equal to number of **input**'s elements regardless of **input**'s content. So, except for the best case where all the elements of **input** have the same value, the number of iterations reduces. In the case of Figure 1.1 (worst case), the number of iterations reduces 5 times (from 180 to 36). In the case of section 1.2.3, the number of iterations reduces 2 times (from 72 to 36). It also means that the number of execution cycles of the hardware significantly reduces.
- We support the operation **selecting the next element to be considered**, the purpose is to skip the selected elements in the previous iterations, thereby, helping to eliminate redundant iterations, making efficiency optimal. Looking at the old algorithm's Matlab code, we can see that in loop 2, each element of matrix input have one iteration. This creates redundant iterations, making efficiency suboptimal.

1.2.5.2 Trade-offs

- As mentioned above, new algorithm needs more hardware resources than the original one to serve operation expanding the selected area and to store **iterated_idx** to serve the operation calculating to ignore the previously selected elements. However, since each element of iterated_idx is only 1 bit , the circuit size for implementing the new algorithm is not too large compared to the old algorithm.

CHAPTER 2. BLOCK DESIGN

This chapter describes the overall and detailed design of the hardware that implements our Regional Maxima algorithm in CHAPTER 1.

2.1 Top module

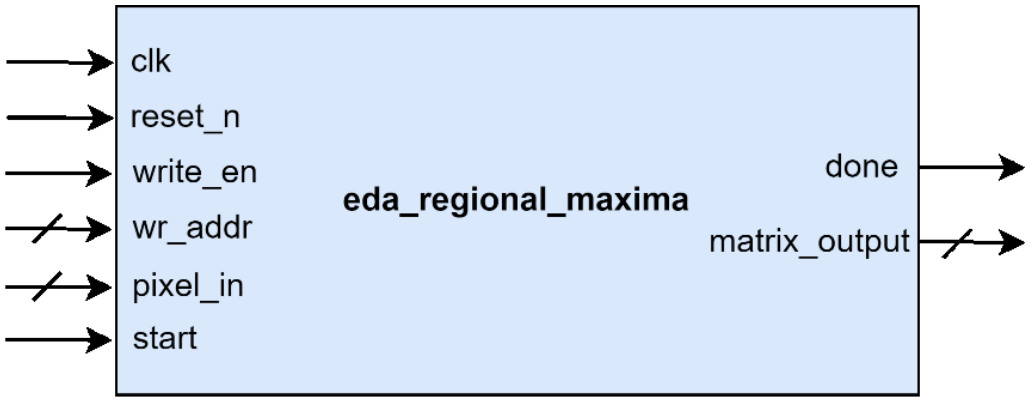

Figure 2.1 eda_regional_maxima block diagram

2.1.1 Port description

Table 2.1 eda_regional_maxima port description

Signal name	Width	I/O	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
write_en	1	Input	Write enable
wr_addr	ADDR_WIDTH	Input	Write address
pixel_in	PIXEL_WIDTH	Input	Pixels' value user want to write to internal image memory of top module in the current cycle
start	1	Input	Start running algorithm after loading image, active HIGH
done	1	Input	Notice that algorithm has been done
matrix_output	M*N	Input	Matrix output – result of the algorithm

Table 2.2 eda_regional_maxima parameter description

Parameter	Value	Description
M	2 to 256 Default: 6	Height of image
N	2 to 256 Default: 6	Width of image
ADDR_WIDTH	$clog2(M) + $clog2(N)	Number of address bits to store image
PIXEL_WIDTH	8 or 16 Default: 8	Number of bits to store a pixel's value

2.1.2 Functional description

eda_regional_maxima is the hardware that implements Regional Maxima algorithm in CHAPTER 1 with an unlimited size and pixel storage bits image, fixed filter window size (3x3). To suit actual use, we support image height from 2 to 256 (pixels), image width from 2 to 256 (pixels) and pixel size of 8 or 16 bits.

Before running algorithm, it is necessary to run the image loading process. This process helps us to store the pixel values taken from signal **pixel_in** to address **wr_addr** of top module's internal image memory. Note that in one cycle, only one pixel (8 bits or 16 bits) is loaded to the image memory. Such design is intended to match the actual RAM where in each read/write command the number of data bits read/write is only 1/4/8/16/32 bits.

After image has already been loaded, user can activate signal **start** to tell the **eda_regional_maxima** to start running the algorithm. Note that if image has not been completely loaded yet, activate signal start still cause **eda_regional_maxima** start running algorithm. So, it will cause fault result. Make sure start is active only when image has already been loaded.

After exactly M*N cycles when **start** is active, **done** is HIGH. This means the algorithm has been done and the value of **matrix_output** is now the true result. Note that if **done** is LOW (algorithm has not been done), **matrix_output** still has value but that is not meaningful (not valid). Once the result of the old image is valid, user can continue to load new images and run Regional Maxima algorithm with new images.

2.1.3 Architecture

Figure 2.2 is **eda_regional_maxima**'s architecture.

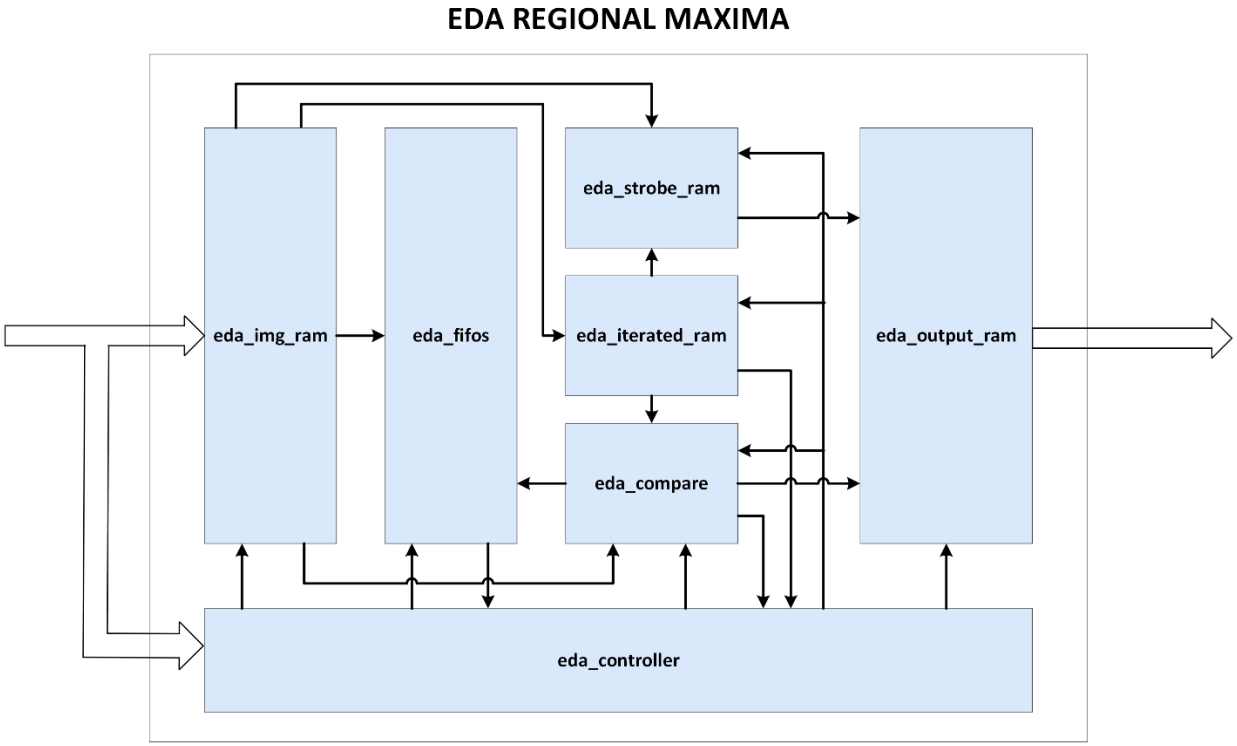

Figure 2.2 eda_regional_maxima architecture

eda_regional_maxima is designed with the following sub-modules:

- **eda_controller** is an FSM used to control all blocks by the following opertations:
  - Write new value to all RAMs except for **eda_img_ram**.
  - Clear values in all RAMs except for **eda_output_ram**.
  - Signal to other modules when to start and finish the execution of the algorithm.
  - Expand or stop expanding selected area then switch to the next iteration.
  - Calculate the next selected address (center_addr).
  - Determine when to read and when to stop reading FIFOs.
- **eda_compare** is responsible for checking if the selected elements are greater than all the neighboring elements of them and checking if any neighboring elements are equal to the selected elements.
- **eda_fifos** contains 8 fifos, each fifo stores the position of one neighboring pixel relative to the currently selected pixel: up left, up, up right, left, right, down left, down, down right if they have the same value as the selected pixel.

- **eda_img_ram** is the internal memory which is used to store input image's content and provide the neighboring addresses of the selected image, determining which positions on the 3x3 window are the valid positions (not out of matrix **input**).
- **eda_iterated_ram** is the internal memory which is used to store matrix **iterated_idx** as decribed in CHAPTER 1 and perform the operation **selecting the next element to be considered** in CHAPTER 1.
- **eda_strobe_ram** is the internal memory which is used to store matrix **strobe** as decribed in CHAPTER 1 provide and provide the currently selected positions to other modules.
- **eda_output_ram** updates the matrix output based on strobed signal.

2.1.4 Timing waveforms

Figure 2.3 is the timing waveform of **eda_regional_maxima**.

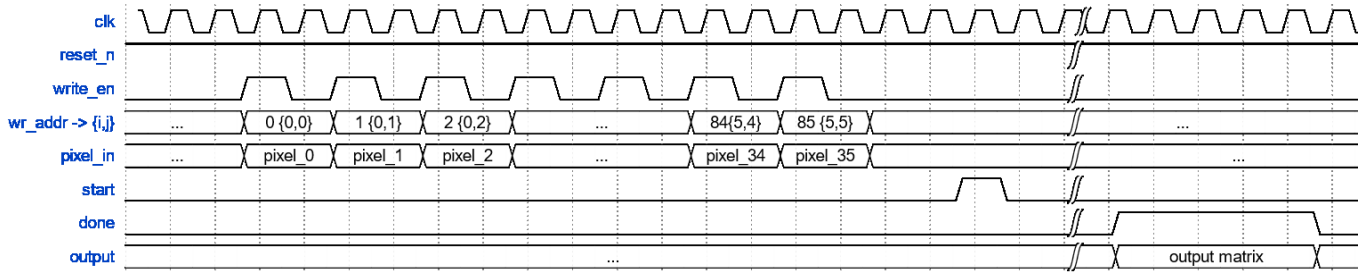

Figure 2.3 eda_regional_maxima timing waveform

2.2 Sub module

2.2.1 eda_controller

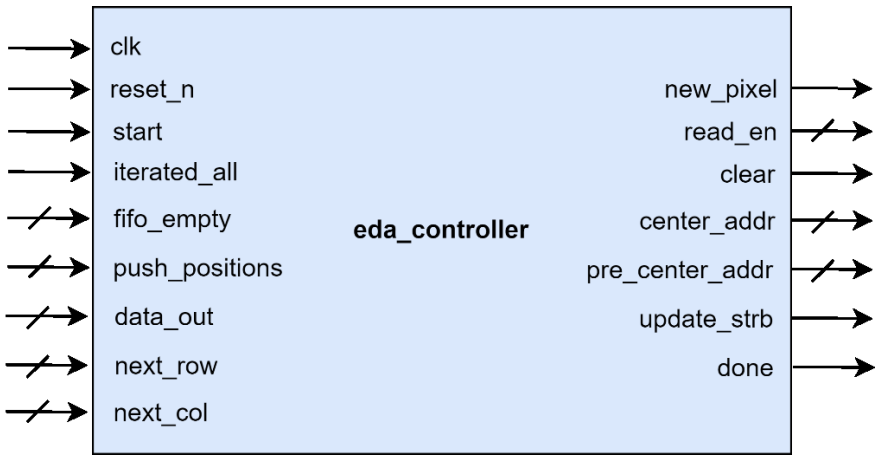

Figure 2.4 eda_controller block diagram

2.2.1.1 Port description

Table 2.3 eda_controller port description

Signal name	Width	I/O	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
start	1	Input	Start running algorithm after loading image, active HIGH
iterated_all	1	Input	Notice that all pixels are iterated
fifo_empty	WINDOW_WIDTH - 1	Input	FIFOs empty flags
push_positions	WINDOW_WIDTH - 1	Input	Positions need to be pushed into FIFOs
data_out	ADDR_WIDTH	Input	Address retrieved from FIFOs
next_row	I_WIDTH	Input	If selected area is not needed to expand, this is the next row of next iteration
next_col	J_WIDTH	Input	If selected area is not needed to expand, this is the next column of next iteration
new_pixel	1	Output	Flag signals that there is a new pixel is selected if the algorithm has not been done
read_en	WINDOW_WIDTH	Output	FIFOs read enable
clear	1	Output	Clear RAMs (eda_img_ram, eda_iterated_ram and eda_strobe_ram)
center_addr	ADDR_WIDTH	Output	The next selected address
pre_center_addr	ADDR_WIDTH	Output	Predicted value of the next selected address (same value but 1 cycle earlier than center_addr)
update_strb	1	Output	Flag update strobe
done	1	Output	Flag done (algorithm has been done)

Table 2.4 eda_controller parameter description

Parameter	Value	Description
WINDOW_WIDTH	9	Number of pixels in the area of a filter window
ADDR_WIDTH	$clog2(M) + $clog2(N)	Number of address bits to store image
I_WIDTH	$clog2(M)	Number of bits to store **input**'s rows
J_WIDTH	$clog2(N)	Number of bits to store **input**'s columns

2.2.1.2 Functional description

eda_controller is an FSM used to control all blocks by the following opertations:

- Write new value to all RAMs except for **eda_img_ram**.
- Clear values in all RAMs except for **eda_output_ram**.
- Signal to other modules when to start and finish the execution of the algorithm.
- Expand or stop expanding selected area then switch to the next iteration.
- Calculate the next selected address (center_addr).
- Determine when to read and when to stop reading FIFOs.

2.2.1.3 Finite State Machine

Figure 2.5 is the FSM of **eda_controller** with some default assignments and reset values described in Table 2.5.

Table 2.5 eda_controller internal signals description

Signal	Mode	Type	Scheme	Default	Reset
clear	output	reg	Combinational	0	
update_strb	output	reg	Combinational	0	
new_pixel	output	reg	Clocked		0
done	output	reg	Clocked		0
update_addr	local	reg	Combinational	0	
extend_addr	local	reg	Combinational	0	
check_next	local	wire	Combinational	0	

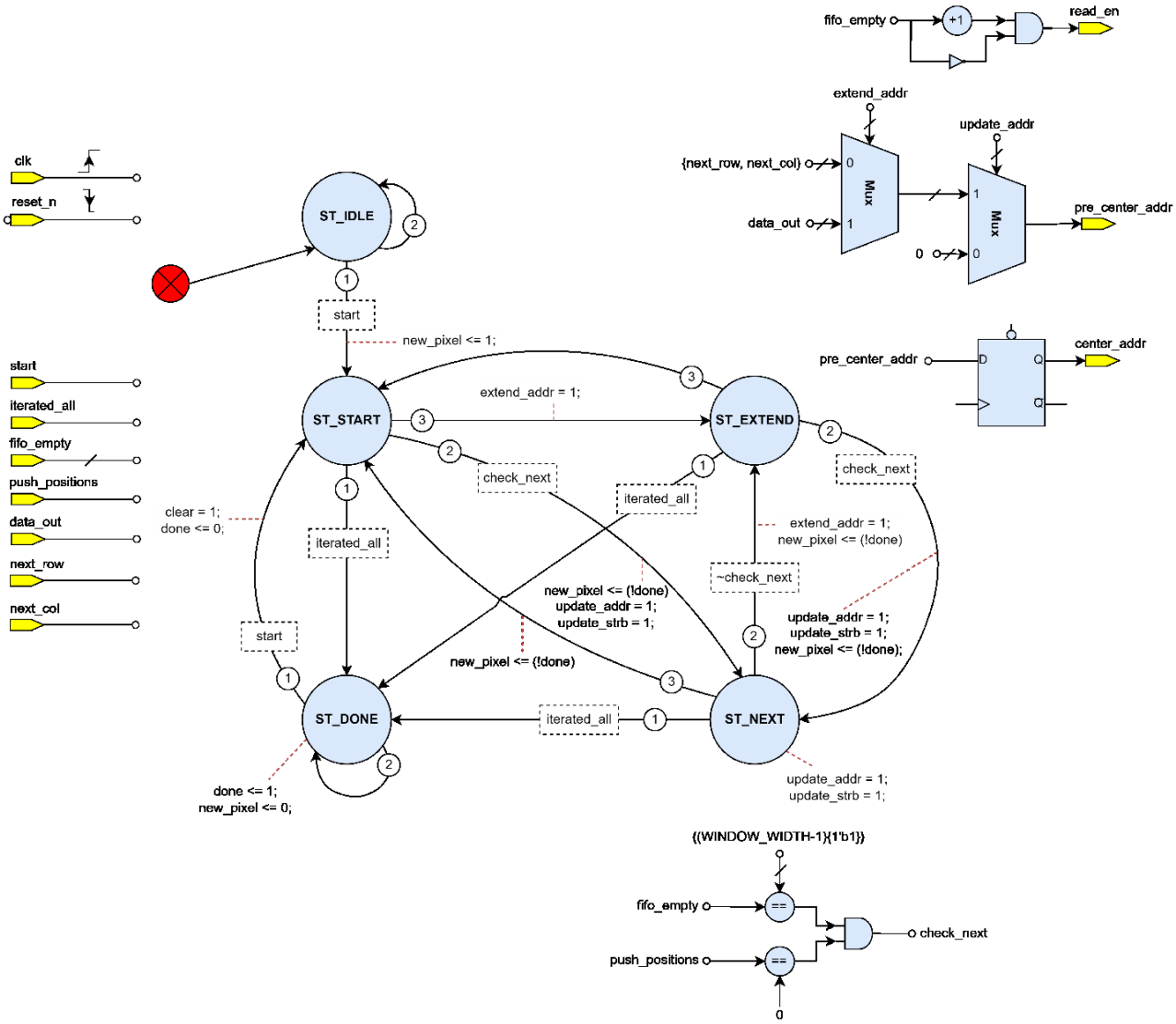

Figure 2.5 eda_controller finite state machine

2.2.2 eda_compare

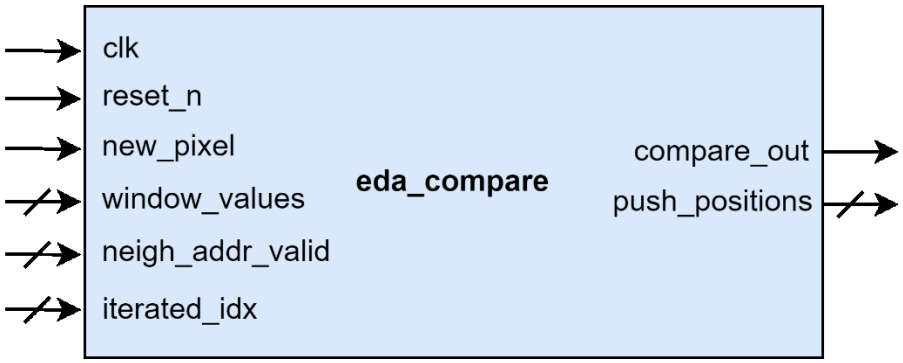

Figure 2.6 eda_compare block diagram

2.2.2.1 Port description

Table 2.6 eda_compare port description

Signal name	Width	I/O	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
new_pixel	1	Input	Flag signals that there is a new pixel is selected if the algorithm has not been done
window_values	PIXEL_WIDTH * WINDOW_WIDTH	Input	Concatenation values of all pixels within the 3x3 window area
neigh_addr_valid	WINDOW_WIDTH - 1	Input	Indicates which positions on the window are the valid positions (not out of matrix **input**)
iterated_idx	WINDOW_WIDTH - 1	Input	The positions on the 3x3 window have already been selected before
compare_out	1	Output	Compare result
push_positions	WINDOW_WIDTH	Output	Positions need to be pushed into FIFOs

Table 2.7 eda_compare parameter description

Parameter	Value	Description
WINDOW_WIDTH	9	Number of pixels in the area of a filter window
PIXEL_WIDTH	8 or 16 Default: 8	Number of bits to store a pixel's value

2.2.2.2 Functional description

eda_compare is responsible for checking if the selected elements are greater than all the neighboring elements of them. If true, **compare_out** is HIGH. If not, **compare_out** is LOW and **eda_compare** has to check if any neighboring elements are equal to the selected elements. If there are, which are they? The result will be shown on

push_positions. If **push_positions[i]** is HIGH, this means neighboring element at position **i** is equal to the selected element.

2.2.3 eda_fifos

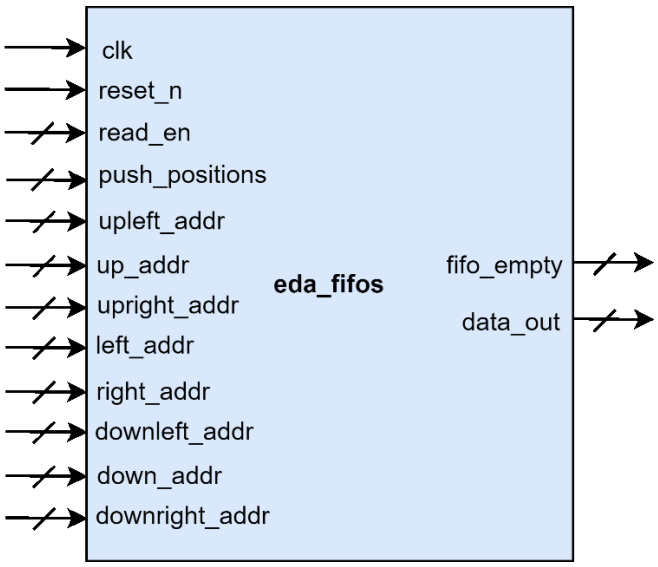

Figure 2.7 eda_fifos block diagram

2.2.3.1 Port description

Table 2.8 eda_fifos port description

Signal name	Width	Input/Output	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
push_positions	WINDOW_WIDTH – 1	Input	Positions need to be pushed into FIFOs
read_en	WINDOW_WIDTH – 1	Input	FIFOs read enable signal
upleft_addr	ADDR_WIDTH	Input	Up left address
up_addr	ADDR_WIDTH	Input	Up address
upright_addr	ADDR_WIDTH	Input	Up right address
left_addr	ADDR_WIDTH	Input	Left address
right_addr	ADDR_WIDTH	Input	Right address
downleft_addr	ADDR_WIDTH	Input	Down left address

down_addr	ADDR_WIDTH	Input	Down address
downright_addr	ADDR_WIDTH	Input	Down right address
fifo_empty	WINDOW_WIDTH – 1	Output	FIFOs empty flags
data_out	ADDR_WIDTH – 1	Output	Address retrieved from FIFO

Table 2.9 eda_fifos parameter description

Parameter	Value	Description
WINDOW_WIDTH	9	Number of pixels in the area of a filter window
ADDR_WIDTH	$clog2(M) + $clog2(N)	Number of address bits to store image
I_WIDTH	$clog2(M)	Number of bits to store **input**'s rows
J_WIDTH	$clog2(N)	Number of bits to store **input**'s columns

2.2.3.2 Functional description

eda_fifos contains 8 fifos, each fifo stores the position of one neighboring pixel relative to the currently selected pixel: up left, up, up right, left, right, down left, down, down right.

If the 8 neighboring pixels have the same value as the currently selected pixel and it hasn't been checked in previous rounds, it will be pushed into the fifo.

2.2.4 eda_img_ram

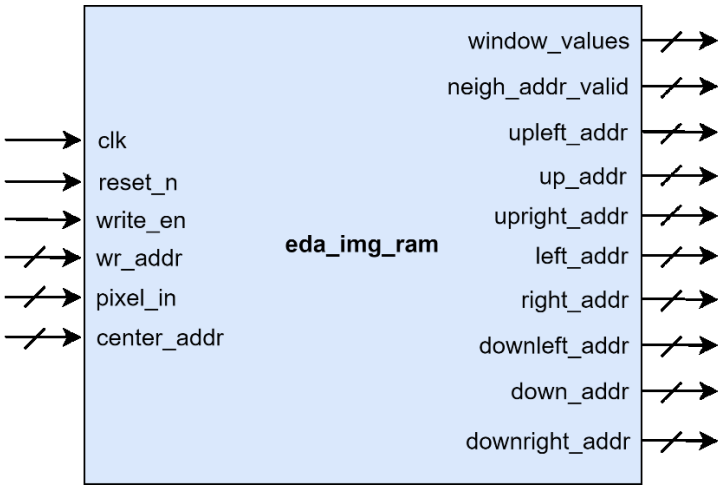

Figure 2.8 eda_img_ram block diagram

2.2.4.1 Port description

Table 2.10 eda_img_ram port description

Signal name	Width	I/O	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
write_en	1	Input	Write enable
wr_addr	ADDR_WIDTH	Input	Write address
pixel_in	PIXEL_WIDTH	Input	Pixel value that user want to write to image RAM
center_addr	ADDR_WIDTH	Input	The next selected address
window_values	PIXEL_WIDTH * WINDOW_WIDTH	Output	Concatenation values of all pixels within the 3x3 window area
neigh_addr_valid	WINDOW_WIDTH - 1	Output	Indicates which positions on the window are the valid positions (not out of matrix **input**)
upleft_addr	ADDR_WIDTH	Output	Up left address
up_addr	ADDR_WIDTH	Output	Up address
upright_addr	ADDR_WIDTH	Output	Up right address
left_addr	ADDR_WIDTH	Output	Left address
right_addr	ADDR_WIDTH	Output	Right address
downleft_addr	ADDR_WIDTH	Output	Down left address
down_addr	ADDR_WIDTH	Output	Down address
downright_addr	ADDR_WIDTH	Output	Down right address

Table 2.11 eda_img_ram parameter description

Parameter	Value	Description
ADDR_WIDTH	$clog2(M) + $clog2(N)	Number of address bits to store image

WINDOW_WIDTH	9	Number of pixels in the area of a filter window
PIXEL_WIDTH	8 or 16 Default: 8	Number of bits to store a pixel's value

2.2.4.2 Functional description

eda_img_ram is the internal memory which is used to store input image's content and provide the neighboring addresses of the selected image, determining which positions on the 3x3 window are the valid positions (not out of matrix **input**).

2.2.5 eda_iterated_ram

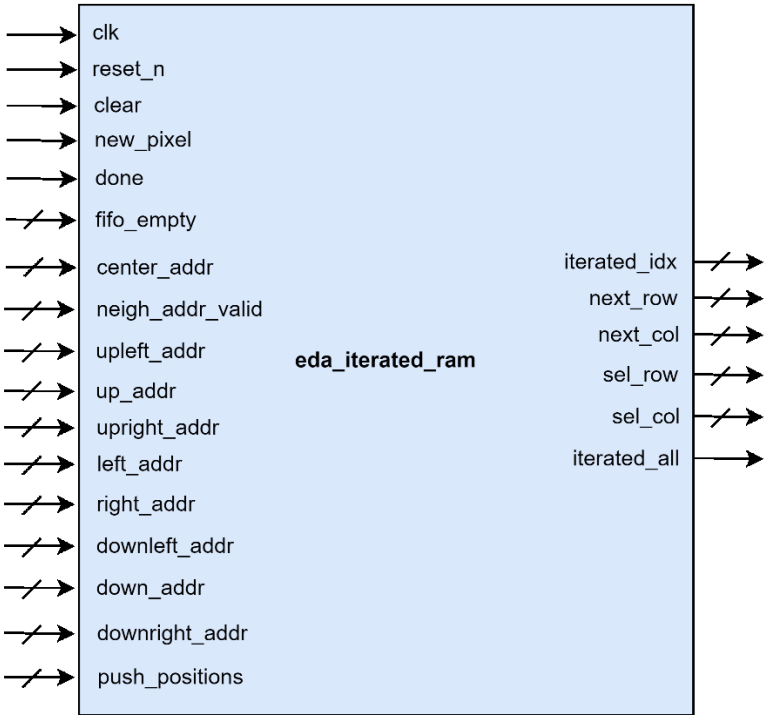

Figure 2.9 eda_iterated_ram block diagram

2.2.5.1 Port description

Table 2.12 eda_iterated_ram port description

Signal name	Width	I/O	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW

clear	1	Input	Clear RAMs (eda_img_ram, eda_iterated_ram and eda_strobe_ram)
new_pixel	1	Input	Flag signals that there is a new pixel is selected if the algorithm has not been done
done	1	Input	Notice that algorithm has been done
fifo_empty	WINDOW_WIDTH - 1	Input	FIFOs empty flags
center_addr	ADDR_WIDTH	Input	The next selected address
upleft_addr	ADDR_WIDTH	Input	Up left address
up_addr	ADDR_WIDTH	Input	Up address
upright_addr	ADDR_WIDTH	Input	Up right address
left_addr	ADDR_WIDTH	Input	Left address
right_addr	ADDR_WIDTH	Input	Right address
downleft_addr	ADDR_WIDTH	Input	Down left address
down_addr	ADDR_WIDTH	Input	Down address
downright_addr	ADDR_WIDTH	Input	Down right address
neigh_addr_valid	WINDOW_WIDTH - 1	Input	Indicates which positions on the window are the valid positions (not out of matrix **input**)
push_positions	WINDOW_WIDTH - 1	Input	Positions need to be pushed into FIFOs
iterated_idx	WINDOW_WIDTH - 1	Output	The positions on the 3x3 window have already been selected before
next_row	I_WIDTH	Output	If selected area is not needed to expand, this is the next row of next iteration
next_col	J_WIDTH	Output	If selected area is not needed to expand, this is the next column of next iteration

sel_row	M	Output	It's **next_row** in the form of one hot code
sel_col	M * N	Output	Predict the next column that will be selected of all rows in the form of one hot code
iterated_all	1	Output	Notice that all pixels are iterated

Table 2.13 eda_iterated_ram parameter description

Parameter	Value	Description
M	2 to 256 Default: 6	Height of image
N	2 to 256 Default: 6	Width of image
I_WIDTH	$clog2(M)	Number of bits to store **input**'s rows
J_WIDTH	$clog2(N)	Number of bits to store **input**'s columns
ADDR_WIDTH	$clog2(M) + $clog2(N)	Number of address bits to store image
WINDOW_WIDTH	9	Number of pixels in the area of a filter window

2.2.5.2 Functional description

eda_iterated_ram is the internal memory which is used to store matrix **iterated_idx** as decribed in CHAPTER 1 and perform the operation **selecting the next element to be considered** in CHAPTER 1. The result of operation **selecting the next element to be considered** will be shown in **next_row**, **next_col**, **sel_row**, **sel_col**.

2.2.6 eda_strobe_ram

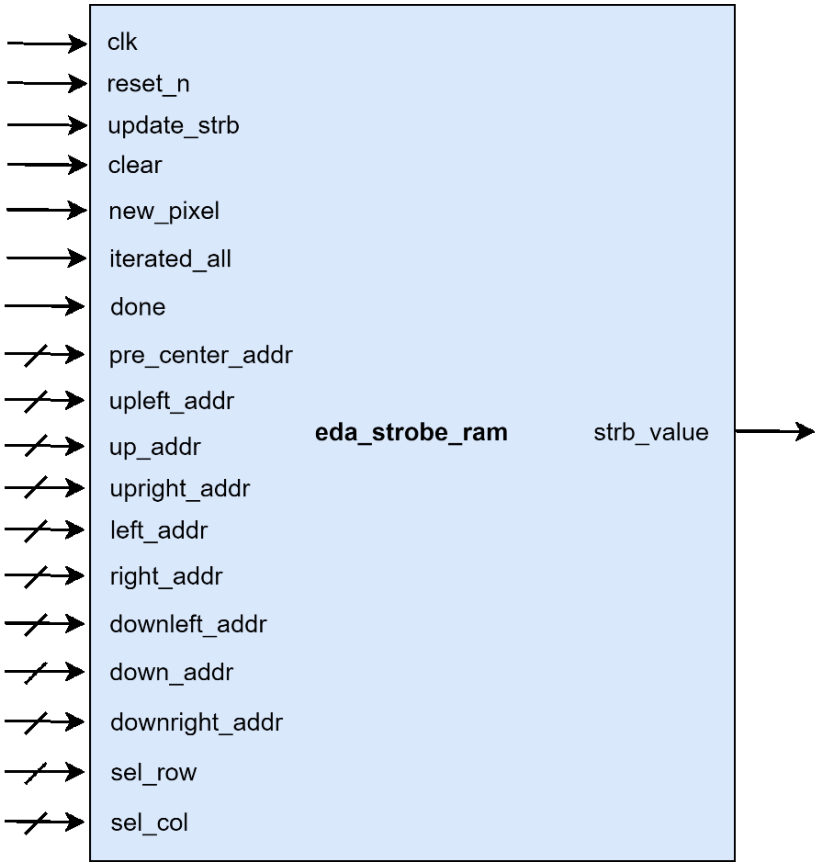

Figure 2.10 eda_strobe_ram block diagram

2.2.6.1 Port description

Table 2.14 eda_strobe_ram port description

Signal name	Width	I/O	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
update_strb	1	Input	
clear	1	Input	Clear RAMs (eda_img_ram, eda_iterated_ram and eda_strobe_ram)
new_pixel	1	Input	Flag signals that there is a new pixel is selected if the algorithm has not been done
iterated_all	1	Input	Notice that all pixels are iterated

done	1	Input	Notice that algorithm has been done
pre_center_addr	ADDR_WIDTH	Input	Predicted value of the next selected address (same value but 1 cycle earlier than center_addr)
upleft_addr	ADDR_WIDTH	Input	Up left address
up_addr	ADDR_WIDTH	Input	Up address
upright_addr	ADDR_WIDTH	Input	Up right address
left_addr	ADDR_WIDTH	Input	Left address
right_addr	ADDR_WIDTH	Input	Right address
downleft_addr	ADDR_WIDTH	Input	Down left address
down_addr	ADDR_WIDTH	Input	Down address
downright_addr	ADDR_WIDTH	Input	Down right address
sel_row	M	Input	It's **next_row** in the form of one hot code
sel_col	M * N	Input	Predict the next column that will be selected of all rows in the form of one hot code
strb_value	M * N	Output	Pixels are being selected

Table 2.15 eda_strobe_ram parameter description

Parameter	Value	Description
M	2 to 256 Default: 6	Height of image
N	2 to 256 Default: 6	Width of image
ADDR_WIDTH	$clog2(M) + $clog2(N)	Number of address bits to store image

2.2.6.2 Functional description

eda_strobe_ram is the internal memory which is used to store matrix **strobe** as decribed in CHAPTER 1 provide and provide the currently selected positions to other modules.

2.2.7 eda_output_ram

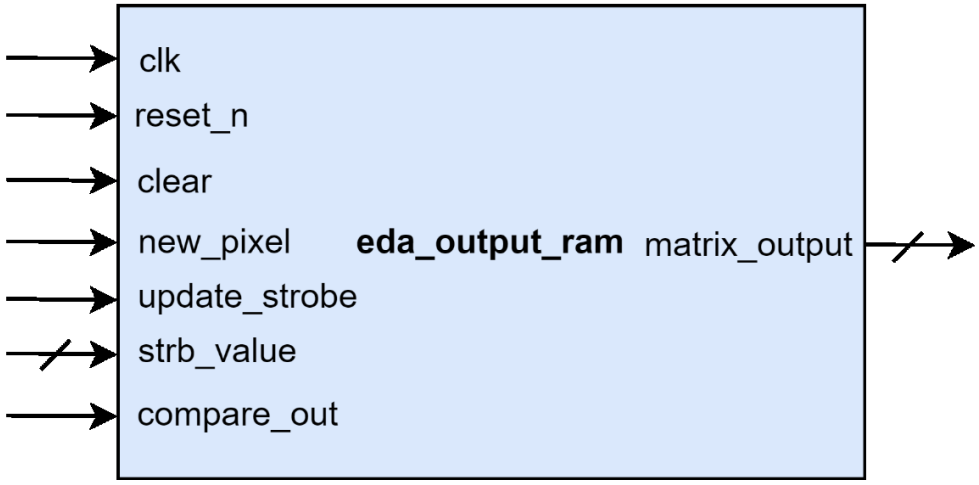

Figure 2.11 eda_output_ram block diagram

2.2.7.1 Port description

Table 2.16 eda_output_ram port description

Signal name	Width	Input/Output	Description
clk	1	Input	Clock signal
reset_n	1	Input	Asynchronous reset, active LOW
clear	1	Input	Synchronous reset, active high, use when load new image
new_pixel	1	Input	Signal that indicate new pixel
update_strobe	1	Input	Signal that pixel updates are being selected
strb_value	M*N	Input	Pixels are being selected
matrix_output	M*N	Output	Matrix output

Table 2.17 eda_output_ram parameter description

Parameter	Value	Description
M	2 to 256 Default: 6	Height of image
N	2 to 256 Default: 6	Width of image

2.2.7.2 Functional description

eda_output_ram updates the matrix output based on strobed signal.

CHAPTER 3. RESULT

This chapter describes the simulation results and synthesis results. We use tools: **Questasim, Synopsys Design Compiler** of **Dolphin Technology Vietnam Center** to simulate and synthesis.

3.1 Simulation result

3.1.1 Software

We code new proposed algorithm by Python (code is located in **software/new_imregion_max.py**) and compare with the results of the original algorithm (original algorithm is converted to Python code and is located in file **software/imregion_max_orginal.py**). After this, we simulate new algorithm with 100000 test cases of image 16x16 (file **software/test.py**). So, the outputs of the new algorithm are equal to the original one.

The results of the software simulation mentioned above are located in folder **software/test_result/**.

3.1.2 Hardware

We create 20000 random images (size 10x10, each pixel value is between 0 and 5) and write a model with correct output to check the output of hardware code. Finally, the results are all correct.

The results is in the folder **sim/tb/** and the log result is located in **sim/tb/vsim.log.**

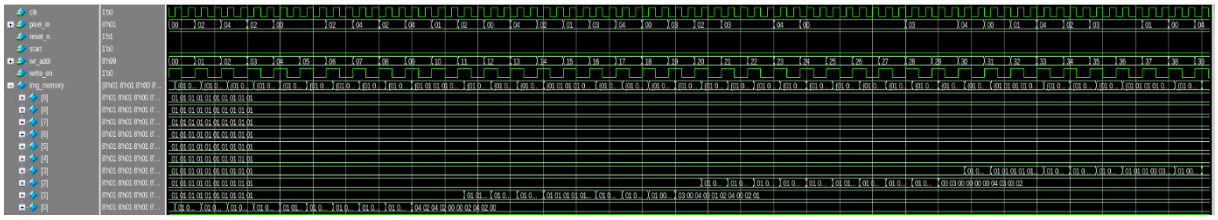

Figure 3.1 Load image into internal memory

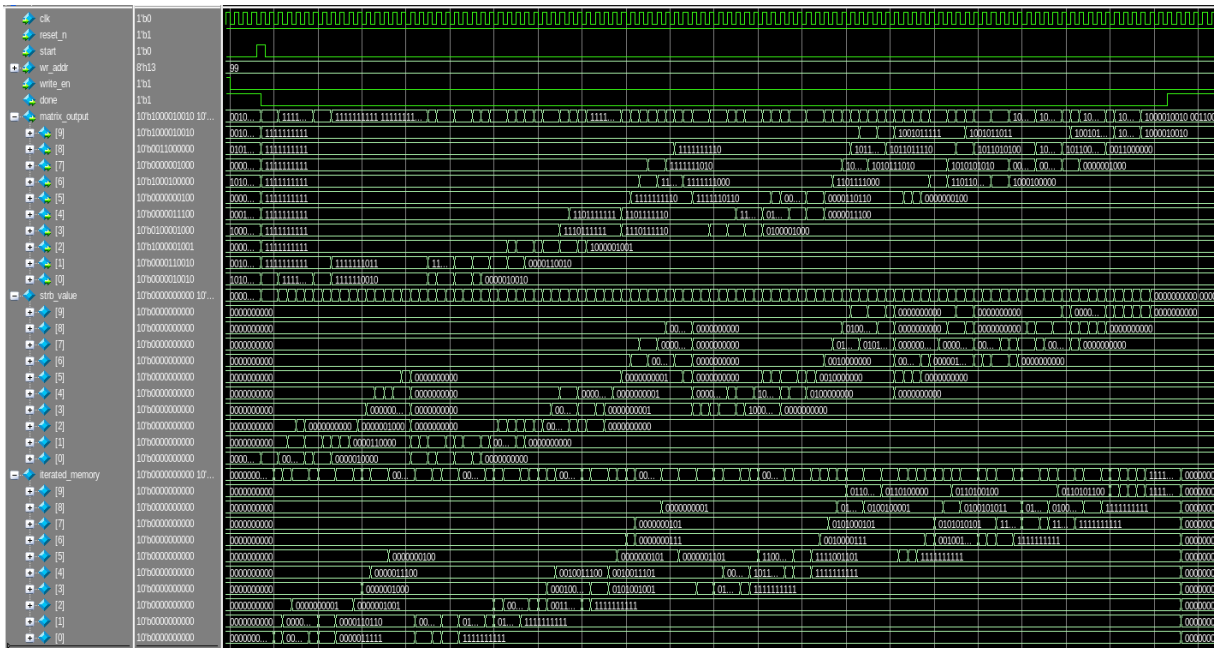

Figure 3.2 Waveform of 1 testcase

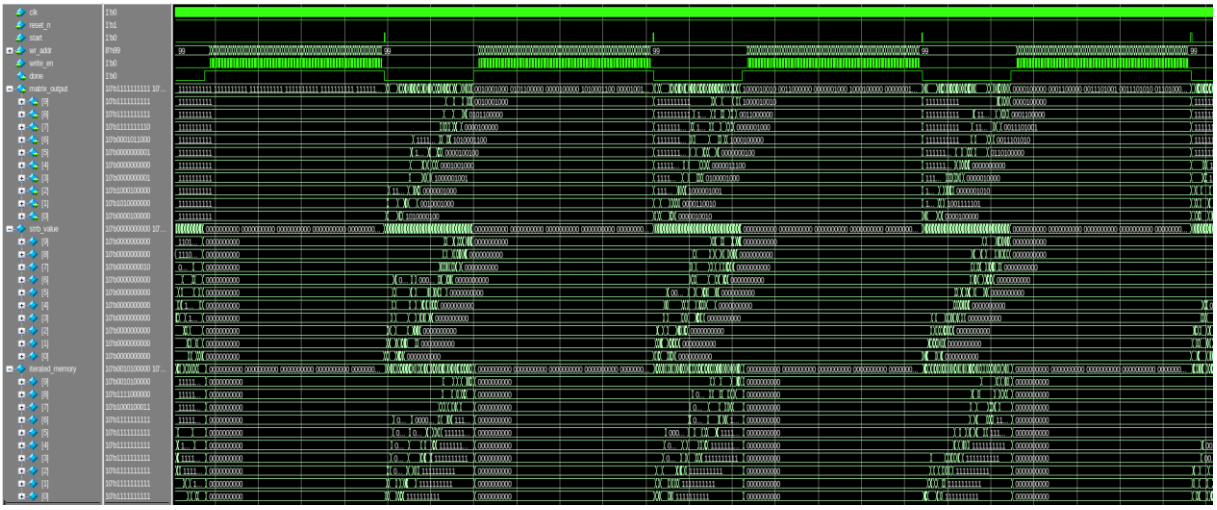

Figure 3.3 Waveform of multiple testcases

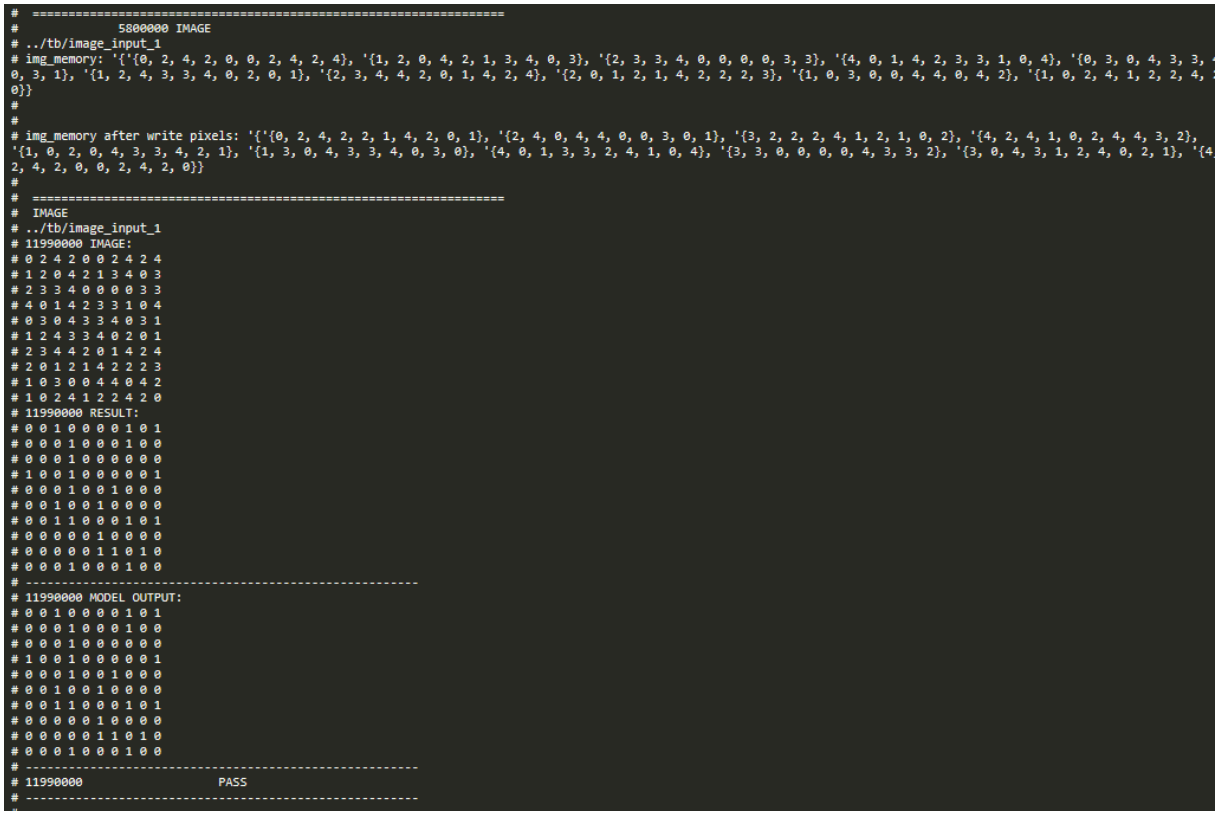

Figure 3.4 Result compared with model

3.2 Synthesis results

We use 28nm library of **Dolphin Technology Vietnam Center** to synthesis and the maximum achievable frequency is 760 MHz.

The result of the synthesis phase is in the folder **syn/reports/**.

Table 3.1 Synthesis results

Timing Path Group 'clk'	---------- Levels of Logic: 16.00000 Critical Path Length: 0.41488 Critical Path Slack: 0.00005 Critical Path Clk Period: 1.31579 Total Negative Slack: 0.00000 No. of Violating Paths: 0.00000 Worst Hold Violation: -0.19615 Total Hold Violation: -65.60418 No. of Hold Violations: 646.00000 ----------
Cell Count	---------- Hierarchical Cell Count: 0 Hierarchical Port Count: 0 Leaf Cell Count: 5523 Buf/Inv Cell Count: 1001 Buf Cell Count: 40 Inv Cell Count: 961 CT Buf/Inv Cell Count: 1 Combinational Cell Count: 4754

	Sequential Cell Count: 769 Macro Count: 0 ----------
Area	---------- Combinational Area: 2719.59799 Noncombinational Area: 1796.04602 Buf/Inv Area: 302.23200 Total Buffer Area: 49.00000 Total Inverter Area: 253.23200 Macro/Black Box Area: 0.00000 Net Area: 0.00000 ---------- Cell Area: 4515.64401 **equivalent to 15359,3333 nand2x1 cells** Design Area: 4515.64401
Power	Global Operating Voltage = 0.81 V Total Dynamic Power = 2.08717 mW (100%) Cell Leakage Power = 313.1223 nW

CHAPTER 4. HDL CODE

4.1 eda_regional_max

Table 4.1 eda_regional_max code

```
`include "eda_global_define.svh"

module eda_regional_max #(
  // synopsys template
  parameter M            = `CFG_M,
  parameter N            = `CFG_N,
  parameter ADDR_WIDTH   = `CFG_ADDR_WIDTH,
  parameter WINDOW_WIDTH = `CFG_WINDOW_WIDTH,
  parameter PIXEL_WIDTH  = `CFG_PIXEL_WIDTH,
  parameter I_WIDTH      = `CFG_I_WIDTH,
  parameter J_WIDTH      = `CFG_J_WIDTH
)
(
  // Port Declarations
  input   wire                          clk,
  input   wire    [PIXEL_WIDTH- 1:0]  pixel_in,
  input   wire                          reset_n,
  input   wire                          start,
  input   wire    [ADDR_WIDTH - 1:0]  wr_addr,
  input   wire                          write_en,
  output  wire                          done,
  output  wire    [M - 1:0][N - 1:0]  matrix_output
);


// Internal Declarations


// Local declarations

// Internal signal declarations
wire  [ADDR_WIDTH - 1:0]                center_addr;
wire                                    clear;
wire                                    update_strb;        // Update strobe
wire                                    compare_out;
wire  [ADDR_WIDTH-1:0]                 data_out;           // Data out from FIFO
wire  [ADDR_WIDTH - 1:0]                down_addr;
wire  [ADDR_WIDTH - 1:0]                downleft_addr;
wire  [ADDR_WIDTH - 1:0]                downright_addr;
wire  [WINDOW_WIDTH - 2:0]              fifo_empty;         // FIFO empty
wire                                    iterated_all;
wire  [WINDOW_WIDTH - 2:0]              iterated_idx;
wire  [ADDR_WIDTH - 1:0]                left_addr;
wire  [WINDOW_WIDTH - 2:0]              neigh_addr_valid;
wire                                    new_pixel;
wire  [J_WIDTH - 1:0]                   next_col;
wire  [I_WIDTH - 1:0]                   next_row;
wire  [WINDOW_WIDTH - 2:0]              push_positions;
wire  [WINDOW_WIDTH - 2:0]              read_en;
wire  [ADDR_WIDTH - 1:0]                right_addr;
```

```
wire  [M - 1:0][N - 1:0]                 strb_value;
wire  [ADDR_WIDTH - 1:0]                 up_addr;
wire  [ADDR_WIDTH - 1:0]                 upleft_addr;
wire  [ADDR_WIDTH - 1:0]                 upright_addr;
wire  [PIXEL_WIDTH * WINDOW_WIDTH - 1:0] window_values;
wire  [ADDR_WIDTH-1:0]                   pre_center_addr;     // Center address
wire  [M - 1:0]                          sel_row;
wire  [M - 1:0][N - 1:0]                 sel_col;



// Instances
eda_compare eda_compare(
  .clk              (clk),
  .reset_n          (reset_n),
  .new_pixel        (new_pixel),
  .window_values    (window_values),
  .neigh_addr_valid (neigh_addr_valid),
  .iterated_idx     (iterated_idx),
  .compare_out      (compare_out),
  .push_positions   (push_positions)
);

eda_controller eda_controller(
  .clk             (clk),
  .reset_n         (reset_n),
  .start           (start),
  .iterated_all    (iterated_all),
  .fifo_empty      (fifo_empty),
  .push_positions  (push_positions),
  .data_out        (data_out),
  .next_row        (next_row),
  .next_col        (next_col),
  .new_pixel       (new_pixel),
  .clear           (clear),
  .center_addr     (center_addr),
  .pre_center_addr (pre_center_addr),
  .update_strb     (update_strb),
  .done            (done),
  .read_en         (read_en)
);

eda_fifos eda_fifos(
  .clk            (clk),
  .reset_n        (reset_n),
  .read_en        (read_en),
  .push_positions (push_positions),
  .upleft_addr    (upleft_addr),
  .up_addr        (up_addr),
  .upright_addr   (upright_addr),
  .left_addr      (left_addr),
  .right_addr     (right_addr),
  .downleft_addr  (downleft_addr),
  .down_addr      (down_addr),
  .downright_addr (downright_addr),
  .fifo_empty     (fifo_empty),
  .data_out       (data_out)
);

eda_img_ram eda_img_ram(
```

```
  .clk              (clk),
  .reset_n          (reset_n),
  .write_en         (write_en),
  .wr_addr          (wr_addr),
  .pixel_in         (pixel_in),
  .center_addr      (center_addr),
  .window_values    (window_values),
  .neigh_addr_valid (neigh_addr_valid),
  .upleft_addr      (upleft_addr),
  .up_addr          (up_addr),
  .upright_addr     (upright_addr),
  .left_addr        (left_addr),
  .right_addr       (right_addr),
  .downleft_addr    (downleft_addr),
  .down_addr        (down_addr),
  .downright_addr   (downright_addr)
);

eda_iterated_ram eda_iterated_ram(
  .clk              (clk),
  .reset_n          (reset_n),
  .clear            (clear),
  .new_pixel        (new_pixel),
  .done             (done),
  .fifo_empty       (fifo_empty),
  .center_addr      (center_addr),
  .upleft_addr      (upleft_addr),
  .up_addr          (up_addr),
  .upright_addr     (upright_addr),
  .left_addr        (left_addr),
  .right_addr       (right_addr),
  .downleft_addr    (downleft_addr),
  .down_addr        (down_addr),
  .downright_addr   (downright_addr),
  .neigh_addr_valid (neigh_addr_valid),
  .push_positions   (push_positions),
  .iterated_idx     (iterated_idx),
  .next_row         (next_row),
  .next_col         (next_col),
  .sel_row          (sel_row),
  .sel_col          (sel_col),
  .iterated_all     (iterated_all)
);

eda_output_ram eda_output_ram(
  .clk           (clk),
  .reset_n       (reset_n),
  .clear         (clear),
  .new_pixel     (new_pixel),
  .update_strb   (update_strb),
  .compare_out   (compare_out),
  .strb_value    (strb_value),
  .matrix_output (matrix_output)
);

eda_strobe_ram eda_strobe_ram(
  .clk              (clk),
  .reset_n          (reset_n),
  .update_strb      (update_strb),
  .clear            (clear),
```

```
  .new_pixel        (new_pixel),
  .iterated_all     (iterated_all),
  .done             (done),
  .pre_center_addr  (pre_center_addr),
  .upleft_addr      (upleft_addr),
  .up_addr          (up_addr),
  .upright_addr     (upright_addr),
  .left_addr        (left_addr),
  .right_addr       (right_addr),
  .downleft_addr    (downleft_addr),
  .down_addr        (down_addr),
  .downright_addr   (downright_addr),
  .sel_row          (sel_row),
  .sel_col          (sel_col),
  .strb_value       (strb_value)
);


endmodule // eda_regional_max
```

4.2 eda_controller

Table 4.2 eda_controller code

```
`include "eda_global_define.svh"
module eda_controller #(
  // synopsys template
  parameter PIXEL_WIDTH  = `CFG_PIXEL_WIDTH,
  parameter WINDOW_WIDTH = `CFG_WINDOW_WIDTH,
  parameter ADDR_WIDTH   = `CFG_ADDR_WIDTH,
  parameter I_WIDTH      = `CFG_I_WIDTH,
  parameter J_WIDTH      = `CFG_J_WIDTH
)
(
  // Port Declarations
  input   wire                          clk,              // Clock signal
  input   wire                            reset_n,          // Asynchronous reset,
active LOW
  input   wire                          start,            // Start
  input   wire                            iterated_all,     // Check if all pixels
have been iterated
  input   wire    [WINDOW_WIDTH-2:0]  fifo_empty,       // FIFO empty
  input   wire    [WINDOW_WIDTH-2:0]  push_positions,   // Positions need to push
to FIFO
  input   wire    [ADDR_WIDTH-1:0]    data_out,         // Data out from FIFO
  input   wire    [I_WIDTH - 1:0]     next_row,         // Next row
  input   wire    [J_WIDTH - 1:0]     next_col,         // Next column
  output  reg                         new_pixel,        // Iterate new pixel
  output  wire    [WINDOW_WIDTH-2:0]  read_en,          // Read FIOFO enable
  output  reg                         clear,            // Clear all RAMs
  output  reg     [ADDR_WIDTH-1:0]    center_addr,      // Center address
  output  reg     [ADDR_WIDTH-1:0]    pre_center_addr,
  output  reg                         update_strb,      // Clear strobe
  output  reg                         done              // Done for an image
);
```

```
// Internal Declarations


// Declare any pre-registered internal signals
reg                    new_pixel_cld;
reg                    done_cld;

// Module Declarations
reg update_addr;  // Update address
reg extend_addr;
wire check_next;  // Check when can jump to {next_row, next_col}
reg pre_new_pixel;

// State encoding
parameter
          ST_IDLE   = 3'd0,
          ST_START  = 3'd1,
          ST_NEXT   = 3'd2,
          ST_DONE   = 3'd3,
          ST_EXTEND = 3'd4;

reg [2:0] current_state, next_state;

//----------------------------------------------------------------
// Next State Block for machine csm
//----------------------------------------------------------------
always @(
  check_next,
  current_state,
  iterated_all,
  start
)
begin : next_state_block_proc
  case (current_state)
    ST_IDLE: begin
      if (start)
        next_state = ST_START;
      else
        next_state = ST_IDLE;
    end
    ST_START: begin
      if (iterated_all)
        next_state = ST_DONE;
      else if (check_next)
        next_state = ST_NEXT;
      else
        next_state = ST_EXTEND;
    end
    ST_NEXT: begin
      if (iterated_all)
        next_state = ST_DONE;
      else if (!check_next)
        next_state = ST_EXTEND;
      else
        next_state = ST_START;
    end
    ST_DONE: begin
      if (start)
        next_state = ST_START;
```

```
      else
        next_state = ST_DONE;
    end
    ST_EXTEND: begin
      if (iterated_all)
        next_state = ST_DONE;
      else if (check_next)
        next_state = ST_NEXT;
      else
        next_state = ST_START;
    end
    default:
      next_state = ST_IDLE;
  endcase
end // Next State Block

//-----------------------------------------------------------------
// Output Block for machine csm
//-----------------------------------------------------------------
always @(
  check_next,
  current_state,
  iterated_all,
  start
)
begin : output_block_proc
  // Default Assignment
  clear = 0;
  update_strb = 0;
  // Default Assignment To Internals
  update_addr = 0;
  extend_addr = 0;
  pre_new_pixel = 0;

  // Combined Actions
  case (current_state)
    ST_START: begin
      if (iterated_all) begin
      end
      else if (check_next) begin
        update_addr = 1;
        update_strb = 1;
      end
      else
        extend_addr = 1;
    end
    ST_NEXT: begin
      update_addr = 1;
      update_strb = 1;
      if (iterated_all) begin
      end
      else if (!check_next) begin
        extend_addr = 1;
        update_strb = 0;
        update_addr = 0;
      end
    end
    ST_DONE: begin
      if (start)
        clear = 1;
```

```
    end
    ST_EXTEND: begin
      extend_addr = 1;
      if (iterated_all) begin
      end
      else if (check_next) begin
        extend_addr = 0;
        update_strb = 1;
        update_addr = 1;
      end
    end
  endcase
end // Output Block

//-----------------------------------------------------------------
// Clocked Block for machine csm
//-----------------------------------------------------------------
always @(
  posedge clk,
  negedge reset_n
)
begin : clocked_block_proc
  if (!reset_n) begin
    current_state <= ST_IDLE;
    // Reset Values
    new_pixel_cld <= 0;
    done_cld <= 0;
  end
  else
  begin
    current_state <= next_state;

    // Combined Actions
    case (current_state)
      ST_IDLE: begin
        if (start)
          new_pixel_cld <= 1;
      end
      ST_START: begin
        if (iterated_all) begin
        end
        else if (check_next)
          new_pixel_cld <= (!done_cld);
        else
          new_pixel_cld <= (!done_cld);
      end
      ST_NEXT: begin
        if (iterated_all) begin
        end
        else if (!check_next)
          new_pixel_cld <= (!done_cld);
      end
      ST_DONE: begin
        done_cld <= 1;
        new_pixel_cld <= 0;
        if (start)
          done_cld <= 0;
      end
      ST_EXTEND: begin
        if (iterated_all) begin
```

```
        end
        else if (check_next)
          new_pixel_cld <= (!done_cld);
      end
    endcase
  end
end // Clocked Block

// Concurrent Statements
// Clocked output assignments
always @(
  new_pixel_cld,
  done_cld
)
begin : clocked_output_proc
  new_pixel = new_pixel_cld;
  done = done_cld;
end
// pre_center_addr
always @(*) begin : proc_pre_center_addr
  pre_center_addr = 0;
  if (extend_addr) begin
    pre_center_addr = data_out;
  end
  else if (update_addr) begin
    pre_center_addr = {next_row, next_col};
  end
end

// center_addr
always @(posedge clk or negedge reset_n) begin : proc_center_addr
  if(~reset_n) begin
    center_addr <= 0;
  end else begin
    center_addr <= pre_center_addr;
  end
end

// read_en
assign read_en = ((fifo_empty + 1) & (~fifo_empty));

// check_next
assign check_next = ((push_positions == 0) & (fifo_empty == {(WINDOW_WIDTH-
1){1'b1}}));
endmodule // eda_controller
```

4.3 eda_compare

Table 4.3 eda_compare code

```
`include "eda_global_define.svh"

module eda_compare #(
  parameter M                = `CFG_M                          ,
  parameter N                = `CFG_N                          ,
  parameter PIXEL_WIDTH      = `CFG_PIXEL_WIDTH                ,
  parameter WINDOW_WIDTH     = `CFG_WINDOW_WIDTH               ,
```

```
  parameter ADDR_WIDTH       = `CFG_ADDR_WIDTH
)(
  input                                          clk             ,
  input                                          reset_n         ,
  input                                          new_pixel       ,
  input        [PIXEL_WIDTH * WINDOW_WIDTH - 1:0] window_values   ,
  input        [WINDOW_WIDTH - 2:0]              neigh_addr_valid,
  input        [WINDOW_WIDTH - 2:0]              iterated_idx    ,
  output logic                                   compare_out     ,
  output       [WINDOW_WIDTH - 2:0]              push_positions
);
  logic [WINDOW_WIDTH - 2:0]               equal_positions;
      //Find max value in 9 pixels
      logic [PIXEL_WIDTH - 1:0] max_value    ;
  logic [PIXEL_WIDTH - 1:0] value_l1[0:3];
  generate
    for (genvar i = 0; i < 8; i = i + 2) begin
      eda_max cl1 (
                           .a   (window_values[i * PIXEL_WIDTH + 7 -: PIXEL_WIDTH]
),
                           .b     (window_values[(i  +  1)  *  PIXEL_WIDTH  +  7  -:
PIXEL_WIDTH]),
                           .out (value_l1[i / 2]                                  )
      );
    end
  endgenerate

  logic [PIXEL_WIDTH - 1:0] value_l2[0:1];
  generate
    for (genvar i = 0; i < 4; i = i + 2) begin
      eda_max cl2 (
                           .a   (value_l1[i]    ),
                           .b   (value_l1[i + 1]),
                           .out (value_l2[i / 2])
      );
    end
  endgenerate

  logic [PIXEL_WIDTH - 1:0] value_l3[0:0];

  generate
    for (genvar i = 0; i < 2; i = i + 2) begin
      eda_max cl3 (
                           .a   (value_l2[i]    ),
                           .b   (value_l2[i + 1]),
                           .out (value_l3[i / 2])
      );
    end
  endgenerate

  eda_max clfinal (
             .a   (value_l3[0]                                                  ),
             .b   (window_values[PIXEL_WIDTH * WINDOW_WIDTH - 1 -: PIXEL_WIDTH]),
             .out (max_value                                                    )
  );

  always_comb begin
      if (max_value > window_values[4 * PIXEL_WIDTH + 7 -: PIXEL_WIDTH]) begin
            compare_out = 0;
      end else begin
```

```
      compare_out = 1;
    end
  end

  // Generate equal_positions
  generate
       for (genvar i = 0; i < WINDOW_WIDTH; i++) begin
             if(i < 4) begin
                    always_comb begin
                           if(window_values[i * PIXEL_WIDTH + 7 : i * PIXEL_WIDTH]
== window_values[4 * PIXEL_WIDTH + 7 -: PIXEL_WIDTH]) begin
                                  equal_positions[i] = new_pixel;
                           end else begin
            equal_positions[i] = 0;
          end
                    end
      end else if(i > 4) begin
                    always_comb begin
                           if(window_values[i * PIXEL_WIDTH + 7 : i * PIXEL_WIDTH]
== window_values[4 * PIXEL_WIDTH + 7 -: PIXEL_WIDTH]) begin
                                  equal_positions[i - 1] = new_pixel;
                           end else begin
            equal_positions[i - 1] = 0;
          end
                    end
      end
       end
  endgenerate

  assign push_positions = equal_positions & (~iterated_idx) & neigh_addr_valid;

endmodule
```

4.4 eda_fifos

Table 4.4 eda_fifos code

```
`include "eda_global_define.svh"

module eda_fifos #(
  parameter M                = `CFG_M           ,
  parameter N                = `CFG_N           ,
  parameter PIXEL_WIDTH      = `CFG_PIXEL_WIDTH ,
  parameter WINDOW_WIDTH     = `CFG_WINDOW_WIDTH,
  parameter ADDR_WIDTH       = `CFG_ADDR_WIDTH  ,
  parameter I_WIDTH          = `CFG_I_WIDTH     ,
  parameter J_WIDTH          = `CFG_J_WIDTH     ,
  parameter FIFO_DEPTH       = `CFG_FIFO_DEPTH  ,
  parameter DATA_WIDTH       = `CFG_DATA_WIDTH
)(
  input                                clk             ,
  input                                reset_n         ,
  input        [WINDOW_WIDTH - 2:0]    read_en         ,
  input        [WINDOW_WIDTH - 2:0]    push_positions  ,
  input        [ADDR_WIDTH - 1:0]      upleft_addr     ,
  input        [ADDR_WIDTH - 1:0]      up_addr         ,
  input        [ADDR_WIDTH - 1:0]      upright_addr    ,
  input        [ADDR_WIDTH - 1:0]      left_addr       ,
  input        [ADDR_WIDTH - 1:0]      right_addr      ,
  input        [ADDR_WIDTH - 1:0]      downleft_addr   ,
```

```
  input        [ADDR_WIDTH - 1:0]        down_addr       ,
  input        [ADDR_WIDTH - 1:0]        downright_addr  ,
  output       [WINDOW_WIDTH - 2:0]      fifo_empty      ,
  output       [ADDR_WIDTH - 1:0]        data_out
);

//|----------|--------|-----------|
//| upleft/7 |   up/6 | upright/5 |
//|----------|--------|-----------|
//| left/4   | center |   right/3 |
//|----------|--------|-----------|
//|downleft/2| down/1 |downright/0|
//|----------|--------|-----------|

logic                    inv_clk;
logic                    upleft_read_en;
logic                    up_read_en;
logic                    upright_read_en;
logic                    left_read_en;
logic                    right_read_en;
logic                    downleft_read_en;
logic                    down_read_en;
logic                    downright_read_en;

logic                    upleft_empty;
logic                    up_empty;
logic                    upright_empty;
logic                    left_empty;
logic                    right_empty;
logic                    downleft_empty;
logic                    down_empty;
logic                    downright_empty;

logic                    upleft_push_position;
logic                    up_push_position;
logic                    upright_push_position;
logic                    left_push_position;
logic                    right_push_position;
logic                    downleft_push_position;
logic                    down_push_position;
logic                    downright_push_position;

logic [ADDR_WIDTH - 1:0] upleft_data_out;
logic [ADDR_WIDTH - 1:0] up_data_out;
logic [ADDR_WIDTH - 1:0] upright_data_out;
logic [ADDR_WIDTH - 1:0] left_data_out;
logic [ADDR_WIDTH - 1:0] right_data_out;
logic [ADDR_WIDTH - 1:0] downleft_data_out;
logic [ADDR_WIDTH - 1:0] down_data_out;
logic [ADDR_WIDTH - 1:0] downright_data_out;
logic [ADDR_WIDTH - 1:0] pre_data_out;

assign {upleft_read_en  , up_read_en  , upright_read_en  ,
        left_read_en    ,               right_read_en    ,
        downleft_read_en, down_read_en, downright_read_en } = read_en;

assign {upleft_push_position  , up_push_position  , upright_push_position  ,
        left_push_position    ,                     right_push_position    ,
        downleft_push_position, down_push_position, downright_push_position } =
push_positions;
```

```
assign fifo_empty = {upleft_empty  , up_empty  , upright_empty  ,
                     left_empty    ,             right_empty    ,
                     downleft_empty, down_empty, downright_empty };

assign data_out = (!downright_empty) ? downright_data_out :
                  (!down_empty     ) ? down_data_out      :
                  (!downleft_empty ) ? downleft_data_out  :
                  (!right_empty    ) ? right_data_out     :
                  (!left_empty     ) ? left_data_out      :
                  (!upright_empty  ) ? upright_data_out   :
                  (!up_empty       ) ? up_data_out        :
                  (!upleft_empty   ) ? upleft_data_out    :
                                       pre_data_out        ;

always_comb begin : proc_pre_data_out
  pre_data_out = 0;
  if (downright_push_position) begin
    pre_data_out = downright_addr;
  end
  else if (down_push_position) begin
    pre_data_out = down_addr;
  end
  else if (downleft_push_position) begin
    pre_data_out = downleft_addr;
  end
  else if (right_push_position) begin
    pre_data_out = right_addr;
  end
  else if (left_push_position) begin
    pre_data_out = left_addr;
  end
  else if (upright_push_position) begin
    pre_data_out = upright_addr;
  end
  else if (up_push_position) begin
    pre_data_out = up_addr;
  end
  else if (upleft_push_position) begin
    pre_data_out = upleft_addr;
  end
end

assign inv_clk = (~clk);

sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH                )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH                )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)        )      // Address width
) sync_fifo_upleft (
  .i_clk             (inv_clk                                ), // Clock signal
  .i_rst_n           (reset_n                                ), // Source domain
asynchronous reset (active low)
  .i_valid_s         (upleft_push_position                   ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain          (upleft_addr                             ), // Push data in
FIFO
```

```
  .i_ready_m         (upleft_read_en                            ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                  ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s         (                                          ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull      (                                       ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full            (                                         ), // FIFO full
flag
  .o_valid_m         (                                         ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty      (                                              ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty           (upleft_empty                              ), // FIFO empty
flag
  .o_dataout         (upleft_data_out                          )  // Pop data from
FIFO
);

sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_up (
  .i_clk             (inv_clk                                ), // Clock signal
  .i_rst_n           (reset_n                                ), // Source domain
asynchronous reset (active low)
  .i_valid_s         (up_push_position                       ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain          (up_addr                                ), // Push data in
FIFO
  .i_ready_m         (up_read_en                             ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                  ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s         (                                       ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull      (                                       ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full            (                                         ), // FIFO full
flag
  .o_valid_m         (                                         ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty      (                                              ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty           (up_empty                               ), // FIFO empty
flag
  .o_dataout         (up_data_out                            )  // Pop data from
FIFO
);

sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_upright (
  .i_clk             (inv_clk                                ), // Clock signal
```

```
  .i_rst_n          (reset_n                                    ), // Source domain
asynchronous reset (active low)
  .i_valid_s        (upright_push_position                      ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain         (upright_addr                               ), // Push data in
FIFO
  .i_ready_m        (upright_read_en                            ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                  ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s        (                                           ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull     (                                         ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full            (                                          ), // FIFO full
flag
  .o_valid_m        (                                           ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty     (                                                ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty          (upright_empty                              ), // FIFO empty
flag
  .o_dataout        (upright_data_out                          )  // Pop data from
FIFO
);

sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_left (
  .i_clk            (inv_clk                                    ), // Clock signal
  .i_rst_n          (reset_n                                    ), // Source domain
asynchronous reset (active low)
  .i_valid_s        (left_push_position                         ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain         (left_addr                                  ), // Push data in
FIFO
  .i_ready_m        (left_read_en                               ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                  ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s        (                                           ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull     (                                         ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full            (                                          ), // FIFO full
flag
  .o_valid_m        (                                           ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty     (                                                ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty          (left_empty                                 ), // FIFO empty
flag
  .o_dataout        (left_data_out                             )  // Pop data from
FIFO
```

```
);

sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_right (
  .i_clk             (inv_clk                                ), // Clock signal
  .i_rst_n           (reset_n                                ), // Source domain
asynchronous reset (active low)
  .i_valid_s         (right_push_position                     ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain          (right_addr                             ), // Push data in
FIFO
  .i_ready_m         (right_read_en                          ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                 ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s         (                                       ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull      (                                    ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full             (                                      ), // FIFO full
flag
  .o_valid_m         (                                       ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty      (                                          ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty           (right_empty                            ), // FIFO empty
flag
  .o_dataout         (right_data_out                         )  // Pop data from
FIFO
);


sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_downleft (
  .i_clk             (inv_clk                                ), // Clock signal
  .i_rst_n           (reset_n                                ), // Source domain
asynchronous reset (active low)
  .i_valid_s         (downleft_push_position                 ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain          (downleft_addr                          ), // Push data in
FIFO
  .i_ready_m         (downleft_read_en                       ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                 ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s         (                                       ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull      (                                    ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
```

```
  .o_full            (                                          ), // FIFO full
flag
  .o_valid_m         (                                        ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty      (                                             ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty           (downleft_empty                          ), // FIFO empty
flag
  .o_dataout         (downleft_data_out                      )  // Pop data from
FIFO
);


sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_down (
  .i_clk             (inv_clk                                  ), // Clock signal
  .i_rst_n           (reset_n                                 ), // Source domain
asynchronous reset (active low)
  .i_valid_s         (down_push_position                      ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
  .i_datain          (down_addr                               ), // Push data in
FIFO
  .i_ready_m         (down_read_en                            ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                 ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s         (                                        ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull      (                                      ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full            (                                          ), // FIFO full
flag
  .o_valid_m         (                                         ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty      (                                             ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty           (down_empty                              ), // FIFO empty
flag
  .o_dataout         (down_data_out                           )  // Pop data from
FIFO
);


sync_fifo #(
  .FIFO_DEPTH(FIFO_DEPTH               )    , // FIFO depth
  .DATA_WIDTH(DATA_WIDTH               )    , // Data width
  .ADDR_WIDTH($clog2(FIFO_DEPTH)       )      // Address width
) sync_fifo_downright (
  .i_clk             (inv_clk                                  ), // Clock signal
  .i_rst_n           (reset_n                                 ), // Source domain
asynchronous reset (active low)
  .i_valid_s         (downright_push_position                 ), // Request write
data into FIFO
  .i_almostfull_lvl  ($clog2(FIFO_DEPTH)'(FIFO_DEPTH-2)         ), // The number
of empty memory locations in the FIFO at which the o_almostfull flag is active
```

```
  .i_datain          (downright_addr                          ), // Push data in
FIFO
  .i_ready_m         (downright_read_en                       ), // Request read
data from FIFO
  .i_almostempty_lvl ($clog2(FIFO_DEPTH)'('b10)                ), // The number
of empty memory locations in the FIFO at which the o_almostempty flag is active
  .o_ready_s         (                                        ), // Status write
data into FIFO (if FIFO not full then o_ready_s = 1)
  .o_almostfull      (                                     ), // FIFO almostfull
flag (determined by i_almostfull_lvl)
  .o_full            (                                        ), // FIFO full
flag
  .o_valid_m         (                                     ), // Status read
data from FIFO (if FIFO not empty then o_valid_m = 1)
  .o_almostempty     (                                           ), // FIFO
almostempty flag (determined by i_almostempty_lvl)
  .o_empty           (downright_empty                         ), // FIFO empty
flag
  .o_dataout         (downright_data_out                   )  // Pop data from
FIFO
);


endmodule : eda_fifos
```

4.5 eda_img_ram

Table 4.5 eda_img_ram code

```
`include "eda_global_define.svh"

module eda_img_ram #(
  parameter M                = `CFG_M                        ,
  parameter N                = `CFG_N                        ,
  parameter PIXEL_WIDTH      = `CFG_PIXEL_WIDTH              ,
  parameter WINDOW_WIDTH     = `CFG_WINDOW_WIDTH             ,
  parameter ADDR_WIDTH       = `CFG_ADDR_WIDTH               ,
  parameter I_WIDTH          = `CFG_I_WIDTH                  ,
  parameter J_WIDTH          = `CFG_J_WIDTH
)(
  input                                          clk             ,
  input                                          reset_n         ,
  input                                          write_en        ,
  input        [ADDR_WIDTH - 1:0]                wr_addr         ,
  input        [PIXEL_WIDTH- 1:0]                pixel_in        ,
  input        [ADDR_WIDTH - 1:0]                center_addr     , //{i, j}
  output logic [PIXEL_WIDTH * WINDOW_WIDTH - 1:0] window_values   ,
  output logic [WINDOW_WIDTH - 2:0]               neigh_addr_valid,
  output       [ADDR_WIDTH - 1:0]                upleft_addr     ,
  output       [ADDR_WIDTH - 1:0]                up_addr         ,
  output       [ADDR_WIDTH - 1:0]                upright_addr    ,
  output       [ADDR_WIDTH - 1:0]                left_addr       ,
  output       [ADDR_WIDTH - 1:0]                right_addr      ,
  output       [ADDR_WIDTH - 1:0]                downleft_addr   ,
  output       [ADDR_WIDTH - 1:0]                down_addr       ,
  output       [ADDR_WIDTH - 1:0]                downright_addr
);

  logic [M - 1:0][N - 1:0][PIXEL_WIDTH - 1:0] img_memory;
```

```
  //|----------|--------|-----------|
  //| upleft   |   up   | upright   |
  //|----------|--------|-----------|
  //| left     | center |   right   |
  //|----------|--------|-----------|
  //| downleft | down   | downright |
  //|----------|--------|-----------|


  // Write pixel into memory
  logic [I_WIDTH - 1:0] i_pixel;
  logic [J_WIDTH - 1:0] j_pixel;

  assign i_pixel = wr_addr[ADDR_WIDTH - 1:J_WIDTH];
  assign j_pixel = wr_addr[J_WIDTH - 1:0]         ;

  always_ff @(posedge clk or negedge reset_n) begin
    if(~reset_n) begin
      img_memory <= 0;
    end else if(write_en) begin
      img_memory[i_pixel][j_pixel] <= pixel_in;
    end
  end

  // Generate neighborhood address of center address
  logic [I_WIDTH - 1:0] i_center;
  logic [J_WIDTH - 1:0] j_center;

  logic [I_WIDTH - 1:0] i_center_minus;
  logic [I_WIDTH - 1:0] i_center_plus ;
  logic [J_WIDTH - 1:0] j_center_plus ;
  logic [J_WIDTH - 1:0] j_center_minus;

  assign i_center = center_addr[ADDR_WIDTH - 1:J_WIDTH];
  assign j_center = center_addr[J_WIDTH - 1:0]         ;

  assign i_center_minus = i_center - 1;
  assign i_center_plus  = i_center + 1;
  assign j_center_plus  = j_center + 1;
  assign j_center_minus = j_center - 1;

  assign upleft_addr    = {i_center_minus, j_center_minus};
  assign up_addr        = {i_center_minus, j_center      };
  assign upright_addr   = {i_center_minus, j_center_plus };
  assign left_addr      = {i_center      , j_center_minus};
  assign right_addr     = {i_center      , j_center_plus };
  assign downleft_addr  = {i_center_plus , j_center_minus};
  assign down_addr      = {i_center_plus , j_center      };
  assign downright_addr = {i_center_plus , j_center_plus };

  //  assign  neigh_addr  =  {upleft_addr,  up_addr,  upright_addr,  left_addr,
right_addr, downleft_addr, down_addr, downright_addr};

  // Generate neighborhood address valid
  always_comb begin
    if(i_center == 0) begin
      if(j_center == 0) begin
        neigh_addr_valid = 8'b00001011;
      end else if(j_center == N - 1) begin
        neigh_addr_valid = 8'b00010110;
```

```
      end else begin
        neigh_addr_valid = 8'b00011111;
      end
    end else if(i_center == M - 1) begin
      if(j_center == 0) begin
        neigh_addr_valid = 8'b01101000;
      end else if(j_center == N - 1) begin
        neigh_addr_valid = 8'b11010000;
      end else begin
        neigh_addr_valid = 8'b11111000;
      end
    end else if(j_center == 0) begin
      neigh_addr_valid = 8'b01101011;
    end else if(j_center == N - 1) begin
      neigh_addr_valid = 8'b11010110;
    end else begin
      neigh_addr_valid = 8'b11111111;
    end
  end

  // Assign window values
  logic  [PIXEL_WIDTH * WINDOW_WIDTH - 1:0] window_values_real;
  logic  [WINDOW_WIDTH :0]                  addr_valid        ;

  assign addr_valid = {neigh_addr_valid[7:4], 1'b1, neigh_addr_valid[3:0]};
  assign   window_values_real   =   {img_memory[i_center_minus][j_center_minus],
img_memory[i_center_minus][j_center                                          ],
img_memory[i_center_minus][j_center_plus],
                                img_memory[i_center            ][j_center_minus],
img_memory[i_center     ][j_center   ], img_memory[i_center     ][j_center_plus],
                                img_memory[i_center_plus       ][j_center_minus],
img_memory[i_center_plus    ][j_center             ],    img_memory[i_center_plus
][j_center_plus]};
  generate
    for (genvar i = 0; i < WINDOW_WIDTH; i++) begin
      always_comb begin
        if(addr_valid[i]) begin
          window_values[i  *  PIXEL_WIDTH  +  7  -:  8]  =  window_values_real[i  *
PIXEL_WIDTH + 7 -: 8];
        end else begin
          window_values[i * PIXEL_WIDTH + 7 -: 8] = 0;
        end
      end
    end
  endgenerate
endmodule
```

4.6 eda_iterated_ram

Table 4.6 eda_iterated_ram code

```
`include "eda_global_define.svh"

module eda_iterated_ram  #(
  parameter M                = `CFG_M                        ,
  parameter N                = `CFG_N                        ,
  parameter WINDOW_WIDTH     = `CFG_WINDOW_WIDTH             ,
  parameter ADDR_WIDTH       = `CFG_ADDR_WIDTH               ,
  parameter I_WIDTH          = `CFG_I_WIDTH                  ,
  parameter J_WIDTH          = `CFG_J_WIDTH
```

```
)(
  input                              clk             ,
  input                              reset_n         ,
  input                              clear           ,
  input                              new_pixel       ,
  input                              done            ,
  input        [WINDOW_WIDTH - 2:0] fifo_empty      ,
  input        [ADDR_WIDTH - 1:0]   center_addr     , //{i, j}
  input        [ADDR_WIDTH - 1:0]   upleft_addr     ,
  input        [ADDR_WIDTH - 1:0]   up_addr         ,
  input        [ADDR_WIDTH - 1:0]   upright_addr    ,
  input        [ADDR_WIDTH - 1:0]   left_addr       ,
  input        [ADDR_WIDTH - 1:0]   right_addr      ,
  input        [ADDR_WIDTH - 1:0]   downleft_addr   ,
  input        [ADDR_WIDTH - 1:0]   down_addr       ,
  input        [ADDR_WIDTH - 1:0]   downright_addr  ,
  input        [WINDOW_WIDTH - 2:0] neigh_addr_valid,
  input        [WINDOW_WIDTH - 2:0] push_positions  ,
  output logic [WINDOW_WIDTH - 2:0] iterated_idx    ,
  output logic [I_WIDTH - 1:0]      next_row        ,
  output logic [J_WIDTH - 1:0]      next_col        ,
  output logic [M - 1:0]              sel_row          , // Find next row (Ex:
00010..00)
  output logic [M - 1:0][N - 1:0]   sel_col          , // Find next column (Ex:
00010..00) for all rows
  output logic                      iterated_all
);

  logic                                          inv_clk        ;
  logic                                          have_done      ;
       logic [M - 1:0][N - 1:0]                       iterated_memory;
       logic [WINDOW_WIDTH - 2:0][ADDR_WIDTH - 1:0]  addr_arr       ;
  logic [M - 1:0][N - 1:0]                       current_row    ; // Get data from
all rows of iterated RAM
  logic [M - 1:0]                                combine_row    ; // Check if each
row has pixel that has not iterated yet
  logic [N - 1:0]                                   final_sel_col  ; // Find next
column (Ex: 00010..00)

  assign inv_clk = (~clk);

  // Next row
  eda_one_hot_to_bin #(
    .ONE_HOT_WIDTH (M      ),
    .BIN_WIDTH     (I_WIDTH)
  )
  eda_one_hot_to_bin_next_row (
    .one_hot_code (sel_row ),
    .bin_code     (next_row)
  );

  // Next column
  eda_one_hot_to_bin #(
    .ONE_HOT_WIDTH (N      ),
    .BIN_WIDTH     (J_WIDTH)
  )
  eda_one_hot_to_bin_next_col (
    .one_hot_code (final_sel_col ),
    .bin_code     (next_col      )
  );
```

```
  // Get data from all rows of iterated RAM
  generate
    for (genvar i = 0; i < M; i++) begin
      for (genvar j = 0; j < N; j++) begin
        assign current_row[i][j] = iterated_memory[i][j];
      end
    end
  endgenerate

  // Find next column (Ex: 00010..00)
  generate
    for (genvar i = 0; i < M; i++) begin
      assign sel_col[i] = ((current_row[i] + 1) & (~current_row[i]));
    end
  endgenerate

  // Check if each row has pixel that has not iterated yet
  generate
    for (genvar i = 0; i < M; i++) begin
      assign combine_row[i] = |sel_col[i];
    end
  endgenerate

  // Select row
  assign sel_row = (((~combine_row) + 1) & combine_row);

  // Select column
  always_comb begin
    final_sel_col = 0;
    for (int i = 0; i < M; i++) begin
      if (sel_row[i] == 1) begin
        final_sel_col = sel_col[i];
      end
    end
  end

  // Check if all pixels have been iterated
  always_ff @(posedge clk or negedge reset_n) begin : proc_iterated_all
    if(~reset_n) begin
      iterated_all <= 0;
    end else begin
      iterated_all <= ((final_sel_col == 0) & (sel_row == 0)) & (fifo_empty ==
{(WINDOW_WIDTH-1){1'b1}}) & new_pixel;
    end
  end

      assign  addr_arr  =  {upleft_addr,  up_addr,  upright_addr,  left_addr,
right_addr, downleft_addr, down_addr, downright_addr};

  always_ff @(posedge clk or negedge reset_n) begin : proc_have_done
    if(~reset_n) begin
      have_done <= 0;
    end else begin
      if (!have_done) begin
        have_done <= done;
      end
      else begin
        have_done <= 0;
      end
```

```
    end
  end

      always_ff @(posedge inv_clk or negedge reset_n) begin
            if(~reset_n) begin
                  iterated_memory <= 0;
            end else if(have_done) begin
                  iterated_memory <= 0;
            end else begin
                  for (int i = 0; i < WINDOW_WIDTH - 1; i++) begin
                        if(push_positions[i]) begin
                              iterated_memory[addr_arr[i][ADDR_WIDTH        -
1:J_WIDTH]][addr_arr[i][J_WIDTH - 1:0]] <= 1;
                        end
                  end
                  if (new_pixel) begin
                        iterated_memory[center_addr[ADDR_WIDTH              -
1:J_WIDTH]][center_addr[J_WIDTH - 1:0]] <= 1;
                  end
            end
      end

      // genvarerate ouput
      generate
            for (genvar i = 0; i < WINDOW_WIDTH - 1; i++) begin
                  always_comb begin
        if (neigh_addr_valid[i]) begin
          iterated_idx[i]     =     iterated_memory[addr_arr[i][ADDR_WIDTH     -
1:J_WIDTH]][addr_arr[i][J_WIDTH - 1:0]];
        end else begin
          iterated_idx[i] = 0;
        end
                  end
            end
      endgenerate
endmodule
```

4.7 eda_strobe_ram

Table 4.7 eda_strobe_ram code

```
`include "eda_global_define.svh"

module eda_strobe_ram #(
  parameter M                = `CFG_M                        ,
  parameter N                = `CFG_N                        ,
  parameter WINDOW_WIDTH     = `CFG_WINDOW_WIDTH             ,
  parameter ADDR_WIDTH       = `CFG_ADDR_WIDTH               ,
  parameter I_WIDTH          = `CFG_I_WIDTH                  ,
  parameter J_WIDTH          = `CFG_J_WIDTH
) (
  input                       clk             ,
  input                       reset_n         ,
  input                       update_strb     ,
  input                       clear           ,
  input                       new_pixel       ,
  input                       iterated_all    ,
  input                       done            ,
  input  [ADDR_WIDTH - 1:0]   pre_center_addr ,
  input  [ADDR_WIDTH - 1:0]   upleft_addr     ,
```

```
  input  [ADDR_WIDTH - 1:0]   up_addr         ,
  input  [ADDR_WIDTH - 1:0]   upright_addr    ,
  input  [ADDR_WIDTH - 1:0]   left_addr       ,
  input  [ADDR_WIDTH - 1:0]   right_addr      ,
  input  [ADDR_WIDTH - 1:0]   downleft_addr   ,
  input  [ADDR_WIDTH - 1:0]   down_addr       ,
  input  [ADDR_WIDTH - 1:0]   downright_addr  ,
  input  [M-1:0]              sel_row         , // Find next row (Ex: 00010..00)
  input  [M-1:0][N-1:0]      sel_col        , // Find next column (Ex: 00010..00)
for all rows
  output [M-1:0][N-1:0]       strb_value
);

  logic [M - 1:0][N - 1:0]                      strb_memory;
  logic [WINDOW_WIDTH - 2:0][ADDR_WIDTH - 1:0] addr_arr   ;

  //|----------|--------|-----------|
  //| upleft   |   up   | upright   |
  //|----------|--------|-----------|
  //| left     | center |   right   |
  //|----------|--------|-----------|
  //| downleft | down   | downright |
  //|----------|--------|-----------|

  assign addr_arr = {upleft_addr, up_addr, upright_addr, left_addr, right_addr,
downleft_addr, down_addr, downright_addr};

  generate
    for (genvar i = 0; i < M; i = i + 1) begin
      for (genvar j = 0; j < N; j = j + 1) begin
        assign strb_value[i][j] = strb_memory[i][j];
      end
    end
  endgenerate

  always_ff @(posedge clk or negedge reset_n) begin
    if(~reset_n) begin
      for (int i = 0; i < M; i++) begin
        for (int j = 0; j < N; j++) begin
          if ((i == 0) & (j == 0)) begin
            strb_memory[i][j] <= 1;
          end else begin
            strb_memory[i][j] <= 0;
          end
        end
      end
    end else if (clear) begin
      for (int i = 0; i < M; i++) begin
        for (int j = 0; j < N; j++) begin
          if ((i == 0) & (j == 0)) begin
            strb_memory[i][j] <= 1;
          end else begin
            strb_memory[i][j] <= 0;
          end
        end
      end
    end else if (update_strb & new_pixel & (!iterated_all)) begin
      for (int i = 0; i < M; i++) begin
        for (int j = 0; j < N; j++) begin
          if ((i == 0) & (j == 0)) begin
```

```
            strb_memory[i][j] <= 0;
          end else begin
            if (sel_row[i] & sel_col[i][j]) begin
              strb_memory[i][j] <= 1;
            end
            else begin
              strb_memory[i][j] <= 0;
            end
          end
        end
      end
    end else if ((!update_strb) & new_pixel & (!iterated_all)) begin
      strb_memory[pre_center_addr[ADDR_WIDTH-
1:J_WIDTH]][pre_center_addr[J_WIDTH-1:0]] <= 1;
    end
  end

endmodule
```

4.8 eda_output_ram

Table 4.8 eda_output_ram code

```
`include "eda_global_define.svh"

module eda_output_ram #(
  parameter M                = `CFG_M                          ,
  parameter N                = `CFG_N                          ,
  parameter PIXEL_WIDTH      = `CFG_PIXEL_WIDTH                ,
  parameter WINDOW_WIDTH     = `CFG_WINDOW_WIDTH               ,
  parameter ADDR_WIDTH       = `CFG_ADDR_WIDTH                 ,
  parameter I_WIDTH          = `CFG_I_WIDTH                    ,
  parameter J_WIDTH          = `CFG_J_WIDTH
)(
  input                       clk           ,  // Clock
  input                       reset_n       ,  // Asynchronous reset active low
  input                        clear         ,  // Synchronous reset active low,
use when load new image
  input                       new_pixel     ,
  input                       update_strb   ,
  input                       compare_out   ,
  input        [M-1:0][N-1:0] strb_value    ,
  output logic [M-1:0][N-1:0] matrix_output
);

  logic compare_out_tmp;
  // logic iterated_all_reg;

  always_ff @(posedge clk or negedge reset_n) begin : proc_compare_out_tmp
    if(~reset_n) begin
      compare_out_tmp <= 1;
    end else begin
      if (clear | update_strb) begin
        compare_out_tmp <= 1;
      end
      else if ((!compare_out) & new_pixel) begin
        compare_out_tmp <= 0;
      end
    end
  end
```

```
  genvar i, j;
  generate
    for (i = 0; i < M; i = i+1) begin
      for (j = 0; j < N; j = j+1) begin
        always_ff @(posedge clk or negedge reset_n) begin
          if(~reset_n) begin
            matrix_output[i][j] <= 1'b1;
          end else begin
            // clear matrix, using when load new image
            if (clear) begin
              matrix_output[i][j] <= 1'b1;
            end
            // Update pixels which has strb
            else if (update_strb & new_pixel) begin
              if (strb_value[i][j]) begin
                matrix_output[i][j] <= (compare_out & compare_out_tmp);
              end
            end
            else if (new_pixel) begin
              if (strb_value[i][j]) begin
                matrix_output[i][j] <= compare_out_tmp;
              end
            end
          end
        end
      end
    end
  endgenerate


endmodule
```

4.9 eda_max

Table 4.9 eda_max code

```
`include "eda_global_define.svh"

module eda_max #(
  parameter PIXEL_WIDTH = `CFG_PIXEL_WIDTH
)(
  input        [PIXEL_WIDTH - 1:0] a  ,
  input        [PIXEL_WIDTH - 1:0] b  ,
  output logic [PIXEL_WIDTH - 1:0] out
);

  always_comb begin
    if (a > b)
      out = a;
    else
      out = b;
  end
endmodule
```

4.10 eda_one_hot_to_bin

Table 4.10 eda_one_hot_to_bin code

```
`include "eda_global_define.svh"

module eda_one_hot_to_bin #(
  parameter ONE_HOT_WIDTH = `CFG_N      ,
  parameter BIN_WIDTH     = `CFG_J_WIDTH
)
(
  input        [ONE_HOT_WIDTH - 1:0] one_hot_code, // One-hot code
  output logic [BIN_WIDTH - 1:0]     bin_code      // Binary code
);

  always_comb begin
    bin_code = 0;
    for (int i = ONE_HOT_WIDTH - 1; i >= 0; i--) begin
      if (one_hot_code[i]) begin
        bin_code = i;
      end
    end
  end

endmodule
```

4.11 sync_fifo

Table 4.11 sync_fifo code

```
`include "eda_global_define.svh"

module sync_fifo #(
      parameter FIFO_DEPTH = `CFG_FIFO_DEPTH       , // FIFO depth
      parameter DATA_WIDTH = `CFG_DATA_WIDTH       , // Data width
      parameter ADDR_WIDTH = $clog2(FIFO_DEPTH)      // Address width
)
(
      input                    i_clk            , // Clock signal
      input                     i_rst_n          , // Source domain asynchronous
reset (active low)
      input                   i_valid_s        , // Request write data into FIFO
      input   [ADDR_WIDTH-1:0] i_almostfull_lvl , // The number of empty memory
locations in the FIFO at which the o_almostfull flag is active
      input   [DATA_WIDTH-1:0] i_datain         , // Push data in FIFO
      input                    i_ready_m        , // Request read data from FIFO
      input   [ADDR_WIDTH-1:0] i_almostempty_lvl, // The number of empty memory
locations in the FIFO at which the o_almostempty flag is active
      output                   o_ready_s        , // Status write data into FIFO
(if FIFO not full then o_ready_s = 1)
      output                        o_almostfull      , // FIFO almostfull flag
(determined by i_almostfull_lvl)
      output                   o_full           , // FIFO full flag
      output                    o_valid_m        , // Status read data from FIFO
(if FIFO not empty then o_valid_m = 1)
      output                       o_almostempty     , // FIFO almostempty flag
(determined by i_almostempty_lvl)
      output                   o_empty          , // FIFO empty flag
      output  [DATA_WIDTH-1:0] o_dataout          // Pop data from FIFO
```

```
);


      //=========================================
      //      Internal signals and variables
      //=========================================

      wire [ADDR_WIDTH:0] wr_addr; // Write address (Write pointer)
      wire [ADDR_WIDTH:0] rd_addr; // Read address (Read pointer)
      wire                wr_en  ; // Write enable

      //=========================================
      //        Synchronous FIFO memory
      //=========================================

      sync_fifo_mem sync_fifo_mem_inst (
            .clk    (i_clk                  ),
            // .reset_n(i_rst_n                ),
            .wr_data(i_datain               ),
            .wr_addr(wr_addr[ADDR_WIDTH-1:0]),
            .wr_en  (wr_en                  ),
            .rd_addr(rd_addr[ADDR_WIDTH-1:0]),
            .rd_data(o_dataout              )
      );

      //=========================================
      //             Write control
      //=========================================

      write_control write_control_inst (
            .clk     (i_clk    ),
            .reset_n (i_rst_n  ),
            .wr_valid(i_valid_s),
            .wr_full (o_full   ),
            .wr_en   (wr_en    ),
            .wr_addr (wr_addr  )
      );

      //=========================================
      //             Read control
      //=========================================

      read_control read_control_inst (
            .clk     (i_clk    ),
            .reset_n (i_rst_n  ),
            .rd_ready(i_ready_m),
            .rd_empty(o_empty  ),
            .rd_addr (rd_addr  )
      );

      //=========================================
      //             Comparator
      //=========================================

      comparator comparator_inst (
            .clk              (i_clk           ),
            .reset_n          (i_rst_n         ),
            .i_valid_s        (i_valid_s       ),
            .i_ready_m        (i_ready_m       ),
            .wr_addr          (wr_addr         ),
```

```
            .rd_addr          (rd_addr          ),
            .i_almostfull_lvl (i_almostfull_lvl ),
            .i_almostempty_lvl(i_almostempty_lvl),
            .o_ready_s        (o_ready_s        ),
            .o_almostfull     (o_almostfull     ),
            .o_full           (o_full           ),
            .o_valid_m        (o_valid_m        ),
            .o_almostempty    (o_almostempty    ),
            .o_empty          (o_empty          )
      );

endmodule

`include "eda_global_define.svh"

module comparator #(
      parameter FIFO_DEPTH = `CFG_FIFO_DEPTH       , // FIFO depth
      parameter ADDR_WIDTH = $clog2(`CFG_FIFO_DEPTH)  // Address width
)
(
      input                    clk              , // Clock signal
      input                     reset_n          , // Source domain asynchronous
reset (active low)
      input                    i_valid_s        , // Request write data into FIFO
      input                    i_ready_m        , // Request read data from FIFO
      input  [ADDR_WIDTH:0]    wr_addr          , // Write address
      input  [ADDR_WIDTH:0]    rd_addr          , // Read address
      input  [ADDR_WIDTH-1:0] i_almostfull_lvl , // The number of empty memory
locations in the FIFO at which the o_almostfull flag is active
      input  [ADDR_WIDTH-1:0] i_almostempty_lvl, // The number of empty memory
locations in the FIFO at which the o_almostempty flag is active
      output reg               o_ready_s        , // Status write data into FIFO
(if FIFO not full then o_ready_s = 1)
      output reg            o_almostfull    , // FIFO almostfull flag (determined
by i_almostfull_lvl)
      output reg               o_full           , // FIFO full flag
      output reg                o_valid_m        , // Status read data from FIFO
(if FIFO not empty then o_valid_m = 1)
      output reg                   o_almostempty     , // FIFO almostempty flag
(determined by i_almostempty_lvl)
      output                   o_empty            // FIFO empty flag
);

      //=========================================
      //      Internal signals and variables
      //=========================================

      wire [ADDR_WIDTH:0] num_elements; // Number of elements

      //=========================================
      //            Number of elements
      //=========================================

      assign num_elements = wr_addr + ((~rd_addr) + 1); // Number of elements =
write address - read address

      //=========================================
      //                  Flags
      //=========================================
```

```
	// Flag FIFO almost full
	always @(posedge clk or negedge reset_n) begin : proc_o_almostfull
		if(~reset_n) begin
			o_almostfull <= 0;
		end else if ((num_elements == i_almostfull_lvl - 1) & i_valid_s &
(!i_ready_m)) begin
			o_almostfull <= 1;
		end
		else if ((num_elements == i_almostfull_lvl) & i_ready_m &
(!i_valid_s)) begin
			o_almostfull <= 0;
		end
	end

	// Flag FIFO full and flag valid for reading data
	always @(posedge clk or negedge reset_n) begin : proc_o_full
		if(~reset_n) begin
			o_full <= 0;
			o_ready_s <= 1;
		end else if ((((num_elements == (FIFO_DEPTH-1)) & i_valid_s) |
(num_elements == FIFO_DEPTH)) & (!i_ready_m)) begin
			o_full <= 1;
			o_ready_s <= 0;
		end
		else begin
			o_full <= 0;
			o_ready_s <= 1;
		end
	end

	// Flag FIFO almost empty
	always @(posedge clk or negedge reset_n) begin : proc_o_almostempty
		if(~reset_n) begin
			o_almostempty <= 1;
		end else if (((num_elements == i_almostempty_lvl + 1) & i_ready_m &
(!i_valid_s)) | o_empty) begin
			o_almostempty <= 1;
		end
		else if ((num_elements == i_almostempty_lvl) & i_valid_s &
(!i_ready_m)) begin
			o_almostempty <= 0;
		end
	end

	// Flag FIFO empty
	always @(posedge clk or negedge reset_n) begin : proc_o_empty
		if(~reset_n) begin
			// o_empty <= 1;
			o_valid_m <= 0;
		end else if ((((num_elements == 1) & i_ready_m) | (num_elements ==
0)) & (!i_valid_s)) begin
			// o_empty <= 1;
			o_valid_m <= 0;
		end
		else begin
			// o_empty <= 0;
			o_valid_m <= 1;
		end
	end
```

```
  assign o_empty = (num_elements == 0);

endmodule

`include "eda_global_define.svh"

module read_control #(
      parameter MEM_DEPTH  = `CFG_FIFO_DEPTH      , // Memory depth
      parameter DATA_WIDTH = `CFG_DATA_WIDTH      , // Data width
      parameter ADDR_WIDTH = $clog2(MEM_DEPTH) // Address width
)
(
      input                      clk     , // Clock signal
      input                        reset_n , // Source domain asynchronous reset
(active low)
      input                      rd_ready, // Request read data from FIFO
      input                      rd_empty, // FIFO empty flag
      output reg [ADDR_WIDTH:0] rd_addr   // Read address
);

      //=============================================
      //      Internal signals and variables
      //=============================================

      wire rd_en; // Read enable

      //=============================================
      //                Read address
      //=============================================

      always @(posedge clk or negedge reset_n) begin : proc_rd_addr
            if(~reset_n) begin
                  rd_addr <= 0;
            end
    else begin
      if (rd_en & (rd_addr == ADDR_WIDTH'(MEM_DEPTH-1))) begin
        rd_addr <= 0;
      end
      else if (rd_en) begin
                  rd_addr <= rd_addr + 1;
            end
    end
      end

      //=============================================
      //                 Read enable
      //=============================================

      assign rd_en = rd_ready & (!rd_empty);

endmodule

`include "eda_global_define.svh"

module sync_fifo_mem #(
      parameter MEM_DEPTH  = `CFG_FIFO_DEPTH      , // Memory depth
      parameter DATA_WIDTH = `CFG_DATA_WIDTH      , // Data width
      parameter ADDR_WIDTH = $clog2(MEM_DEPTH) // Address width
)
(
```

```
	input                   clk    , // Clock signal
	// input                   reset_n, // Synchonous reset
	input  [DATA_WIDTH-1:0] wr_data, // Write data
	input  [ADDR_WIDTH-1:0] wr_addr, // Write address
	input                   wr_en  , // Write enable
	input  [ADDR_WIDTH-1:0] rd_addr, // Read address
	output [DATA_WIDTH-1:0] rd_data  // Read data
);

	//============================================
	//      Internal signals and variables
	//============================================

	reg [0:MEM_DEPTH-1][DATA_WIDTH-1:0] fifo_mem;  // FIFO memory
                                                    // Number of elements: MEM_DEPTH
                                                    // Data width of each element:
DATA_WIDTH

	//============================================
	//               Read data
	//============================================

	assign rd_data = fifo_mem[rd_addr];

	//============================================
	//           Write data to memory
	//============================================

	always @(posedge clk) begin : proc_wr_data
		if (wr_en) begin
			fifo_mem[wr_addr] <= wr_data;
		end
	end

endmodule

`include "eda_global_define.svh"

module write_control #(
	parameter MEM_DEPTH  = `CFG_FIFO_DEPTH      , // Memory depth
	parameter DATA_WIDTH = `CFG_DATA_WIDTH      , // Data width
	parameter ADDR_WIDTH = $clog2(MEM_DEPTH) // Address width
)
(
	input                     clk     , // Clock signal
	input                       reset_n , // Source domain asynchronous reset
(active low)
	input                     wr_valid, // Request write data into FIFO
	input                     wr_full , // FIFO full flag
	output                    wr_en   , // Write data
	output reg [ADDR_WIDTH:0] wr_addr   // Write address
);

	//============================================
	//               Write address
	//============================================

	always @(posedge clk or negedge reset_n) begin : proc_wr_addr
		if(~reset_n) begin
			wr_addr <= 0;
```

```
            end
    else begin
      if (wr_en & (wr_addr == ADDR_WIDTH'(MEM_DEPTH-1))) begin
        wr_addr <= '0;
      end
      else if (wr_en) begin
                  wr_addr <= wr_addr + 1;
            end
    end
       end

       //=========================================
       //                Write enable
       //=========================================

       assign wr_en = wr_valid & (!wr_full);

endmodule
```