|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY  **SCHOOL OF ELECTRONIC AND TELECOMMUNICATION**    **LSI CONTEST**  **REGIONAL MAXIMA**  **Team name: EDABK – HUST**  **Instructor:** Assoc. Prof. Nguyen Duc Minh  **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](mailto: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](mailto: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](mailto:quang.nd182736@sis.hust.edu.vn)
  + Size of T-shirt: XL

**TABLE OF CONTENTS**

[TABLE OF FIGURES i](#_Toc125905965)

[TABLE OF TABLES iii](#_Toc125905966)

[CHAPTER 1. NEW PROPOSED ALGORITHM 5](#_Toc125905967)

[1.1 Restrictions of original algorithm 5](#_Toc125905968)

[1.2 New proposed algorithm 6](#_Toc125905969)

[1.2.1 Main idea 6](#_Toc125905970)

[1.2.2 Flow chart 8](#_Toc125905971)

[1.2.3 Example 9](#_Toc125905972)

[1.2.4 Operation selecting the next element to be considered 13](#_Toc125905973)

[1.2.5 Impovements and trade-offs of new algorithm 14](#_Toc125905974)

[CHAPTER 2. BLOCK DESIGN 15](#_Toc125905975)

[2.1 Top module 15](#_Toc125905976)

[2.1.1 Port description 15](#_Toc125905977)

[2.1.2 Functional description 16](#_Toc125905978)

[2.1.3 Architecture 17](#_Toc125905979)

[2.1.4 Timing waveforms 18](#_Toc125905980)

[2.2 Sub module 18](#_Toc125905981)

[2.2.1 eda\_controller 18](#_Toc125905982)

[2.2.2 eda\_compare 21](#_Toc125905983)

[2.2.3 eda\_fifos 23](#_Toc125905984)

[2.2.4 eda\_img\_ram 24](#_Toc125905985)

[2.2.5 eda\_iterated\_ram 26](#_Toc125905986)

[2.2.6 eda\_strobe\_ram 29](#_Toc125905987)

[2.2.7 eda\_output\_ram 31](#_Toc125905988)

[CHAPTER 3. RESULT 33](#_Toc125905989)

[3.1 Simulation result 33](#_Toc125905990)

[3.1.1 Software 33](#_Toc125905991)

[3.1.2 Hardware 33](#_Toc125905992)

[3.2 Synthesis results 35](#_Toc125905993)

[CHAPTER 4. HDL CODE 37](#_Toc125905994)

[4.1 eda\_regional\_max 37](#_Toc125905995)

[4.2 eda\_controller 40](#_Toc125905996)

[4.3 eda\_compare 44](#_Toc125905997)

[4.4 eda\_fifos 46](#_Toc125905998)

[4.5 eda\_img\_ram 53](#_Toc125905999)

[4.6 eda\_iterated\_ram 55](#_Toc125906000)

[4.7 eda\_strobe\_ram 58](#_Toc125906001)

[4.8 eda\_output\_ram 60](#_Toc125906002)

[4.9 eda\_max 61](#_Toc125906003)

[4.10 eda\_one\_hot\_to\_bin 62](#_Toc125906004)

[4.11 sync\_fifo 62](#_Toc125906005)

TABLE OF FIGURES

[Figure 1.1 Matlab result in case of large number of loops 5](#_Toc125903480)

[Figure 1.2 The simplest case of finding regional maxima 6](#_Toc125903481)

[Figure 1.3 The case when the area to be considered is expanded 7](#_Toc125903482)

[Figure 1.4 Proposed algorithm flow chart 8](#_Toc125903483)

[Figure 1.5 Process 1 of the new algorithm 9](#_Toc125903484)

[Figure 1.6 Iterate element in position row 1, column 1 10](#_Toc125903485)

[Figure 1.7 The first step of expanding selected area 10](#_Toc125903486)

[Figure 1.8 The last step of expanding selected area 11](#_Toc125903487)

[Figure 1.9 Process update output 11](#_Toc125903488)

[Figure 1.10 Status of four matrices after the first iteration 12](#_Toc125903489)

[Figure 1.11 The next iteration 12](#_Toc125903490)

[Figure 1.12 The result of the new algorithm 13](#_Toc125903491)

[Figure 1.13 Example of operation selecting the next element to be considered 14](#_Toc125903492)

[Figure 2.1 eda\_regional\_maxima block diagram 15](#_Toc125903493)

[Figure 2.2 eda\_regional\_maxima architecture 17](#_Toc125903494)

[Figure 2.3 eda\_regional\_maxima timing waveform 18](#_Toc125903495)

[Figure 2.4 eda\_controller block diagram 18](#_Toc125903496)

[Figure 2.5 eda\_controller finite state machine 21](#_Toc125903497)

[Figure 2.6 eda\_compare block diagram 21](#_Toc125903498)

[Figure 2.7 eda\_fifos block diagram 23](#_Toc125903499)

[Figure 2.8 eda\_img\_ram block diagram 24](#_Toc125903500)

[Figure 2.9 eda\_iterated\_ram block diagram 26](#_Toc125903501)

[Figure 2.10 eda\_strobe\_ram block diagram 29](#_Toc125903502)

[Figure 2.11 eda\_output\_ram block diagram 31](#_Toc125903503)

[Figure 3.1 Load image into internal memory 33](#_Toc125903504)

[Figure 3.2 Waveform of 1 testcase 34](#_Toc125903505)

[Figure 3.3 Waveform of multiple testcases 34](#_Toc125903506)

[Figure 3.4 Result compare with model 35](#_Toc125903507)

TABLE OF TABLES

[Table 2.1 eda\_regional\_maxima port description 15](#_Toc125903508)

[Table 2.2 eda\_regional\_maxima parameter description 16](#_Toc125903509)

[Table 2.3 eda\_controller port description 19](#_Toc125903510)

[Table 2.4 eda\_controller parameter description 20](#_Toc125903511)

[Table 2.5 eda\_controller internal signals description 20](#_Toc125903512)

[Table 2.6 eda\_compare port description 22](#_Toc125903513)

[Table 2.7 eda\_compare parameter description 22](#_Toc125903514)

[Table 2.8 eda\_fifos port description 23](#_Toc125903515)

[Table 2.9 eda\_fifos parameter description 24](#_Toc125903516)

[Table 2.10 eda\_img\_ram port description 25](#_Toc125903517)

[Table 2.11 eda\_img\_ram parameter description 25](#_Toc125903518)

[Table 2.12 eda\_iterated\_ram port description 26](#_Toc125903519)

[Table 2.13 eda\_iterated\_ram parameter description 28](#_Toc125903520)

[Table 2.14 eda\_strobe\_ram port description 29](#_Toc125903521)

[Table 2.15 eda\_strobe\_ram parameter description 30](#_Toc125903522)

[Table 2.16 eda\_output\_ram port description 31](#_Toc125903523)

[Table 2.17 eda\_output\_ram parameter description 32](#_Toc125903524)

[Table 3.1 Synthesis results 35](#_Toc125903525)

[Table 4.1 eda\_regional\_max code 37](#_Toc125903526)

[Table 4.2 eda\_controller code 40](#_Toc125903527)

[Table 4.3 eda\_compare code 44](#_Toc125903528)

[Table 4.4 eda\_fifos code 46](#_Toc125903529)

[Table 4.5 eda\_img\_ram code 53](#_Toc125903530)

[Table 4.6 eda\_iterated\_ram code 55](#_Toc125903531)

[Table 4.7 eda\_strobe\_ram code 58](#_Toc125903532)

[Table 4.8 eda\_output\_ram code 60](#_Toc125903533)

[Table 4.9 eda\_max code 61](#_Toc125903534)

[Table 4.10 eda\_one\_hot\_to\_bin code 62](#_Toc125903535)

[Table 4.11 sync\_fifo code 62](#_Toc125903536)

# 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.

## 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.

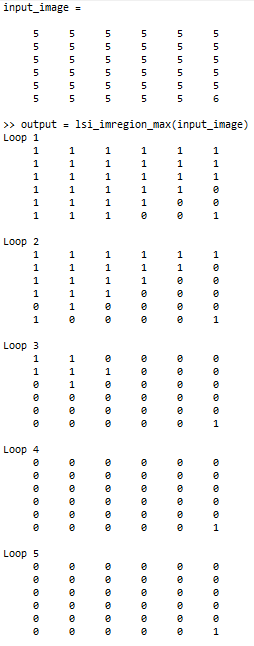


Figure 1.1 Matlab result in case of large number of loops

Figure 1.1 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.

## 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.

### 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.

Table

Description automatically generated

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.

Table

Description automatically generated

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.

### Flow chart

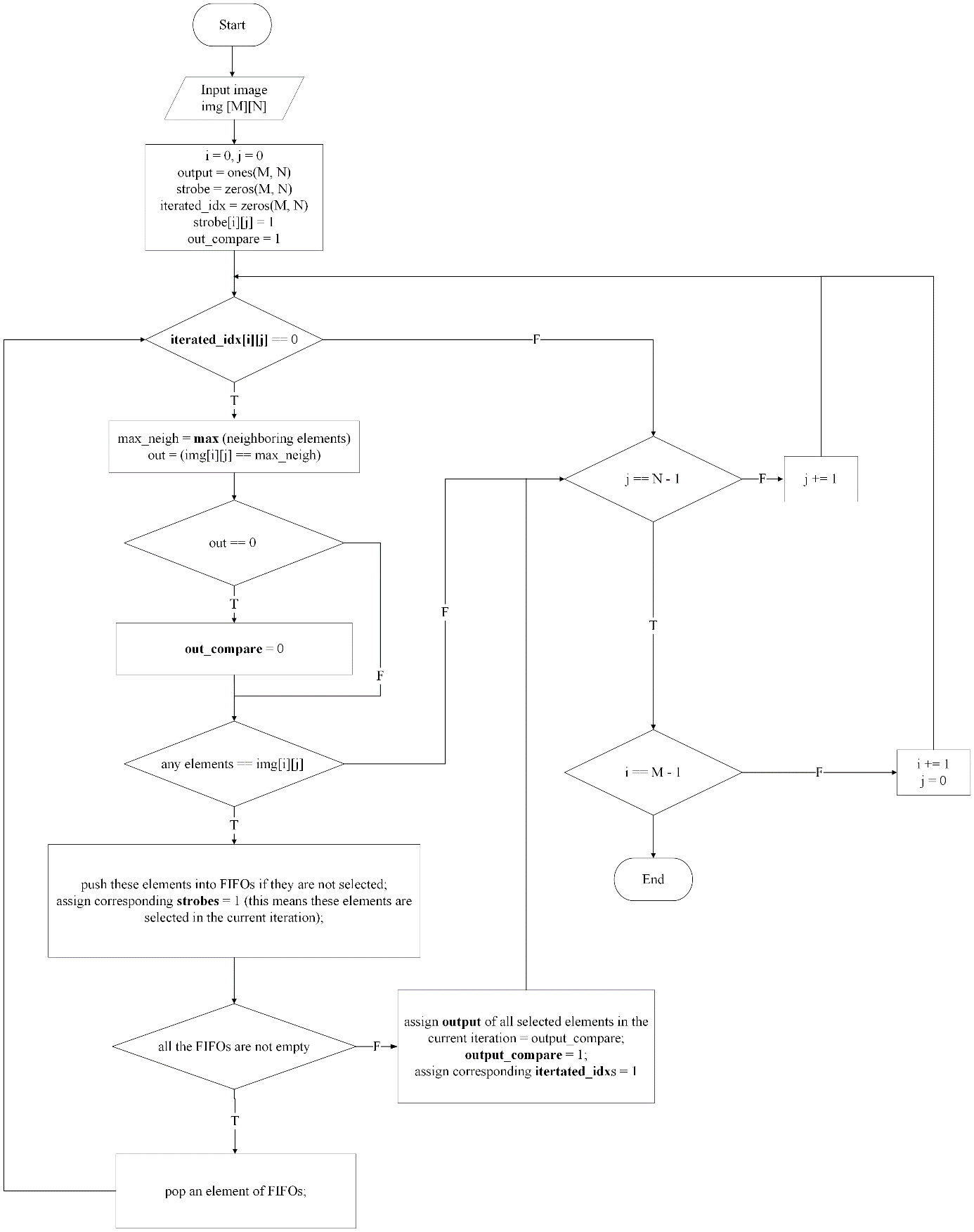


Figure 1.4 Proposed algorithm flow chart

**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.

### 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.

Table

Description automatically generated

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.

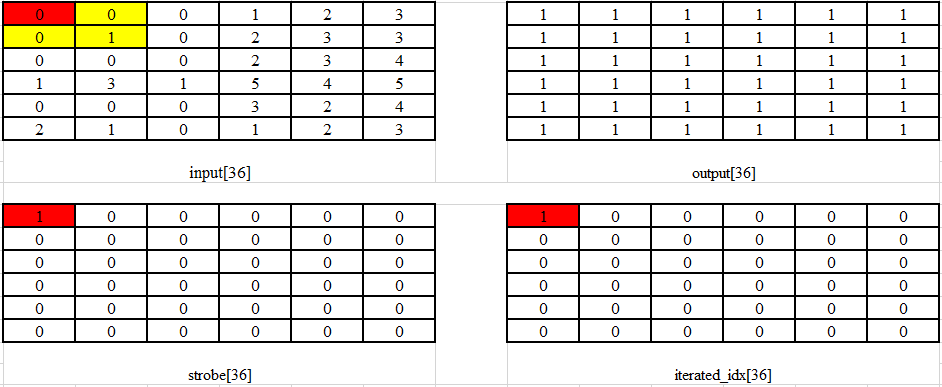


Figure 1.6 Iterate element in position row 1, column 1

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.

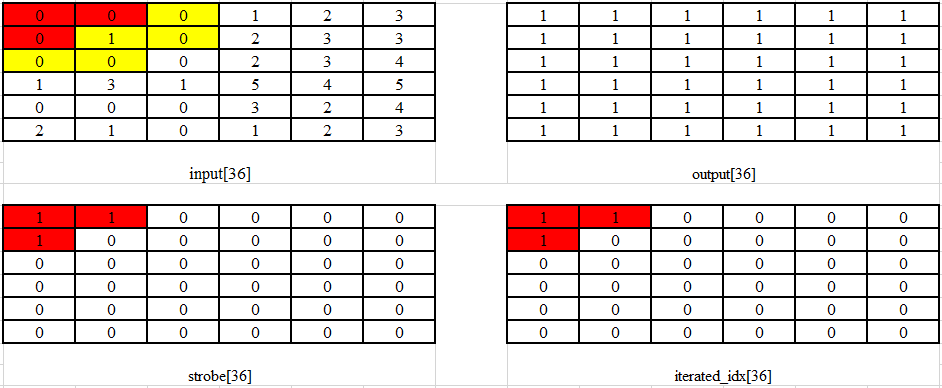


Figure 1.7 The first step of expanding selected area

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.

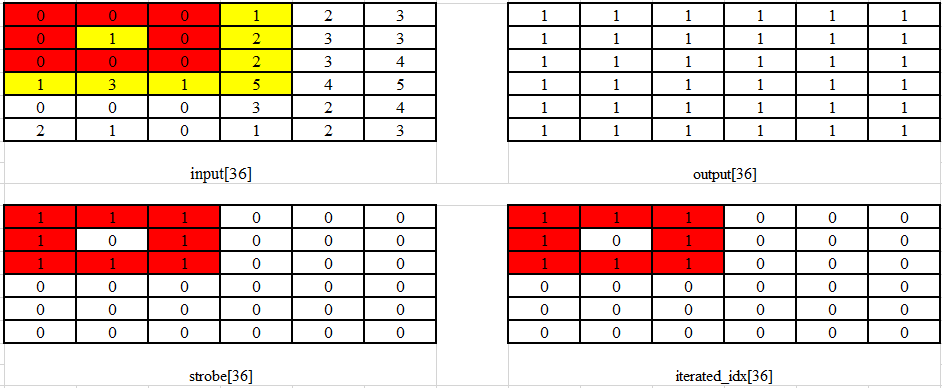


Figure 1.8 The last step of expanding selected area

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.

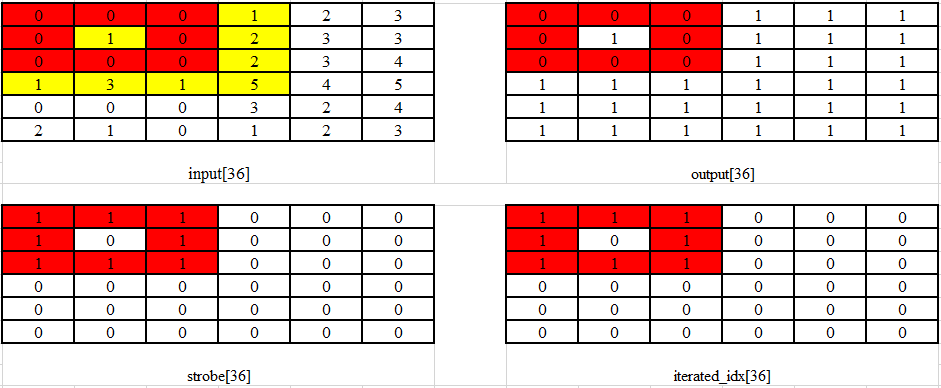


Figure 1.9 Process update output

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

A picture containing text, crossword puzzle

Description automatically generated

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.

Table, calendar

Description automatically generated

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.

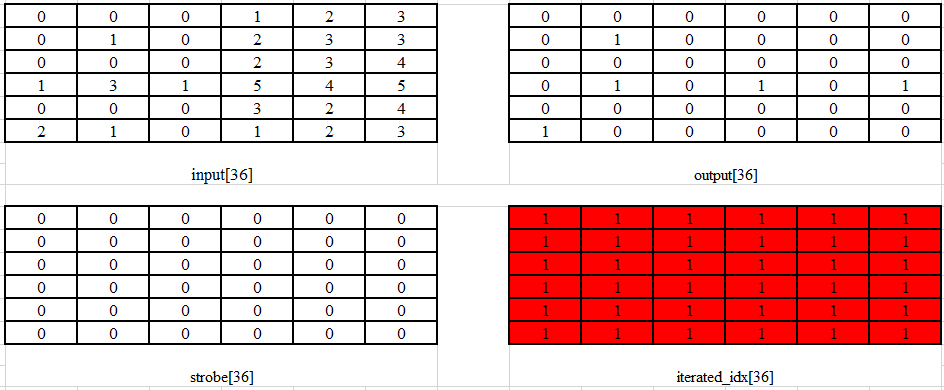


Figure 1.12 The result of the new algorithm

### 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).

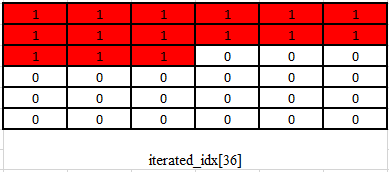


Figure 1.13 Example of operation selecting the next element to be considered

### Impovements and trade-offs of new algorithm

#### 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.

#### 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.

# BLOCK DESIGN

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

## Top module

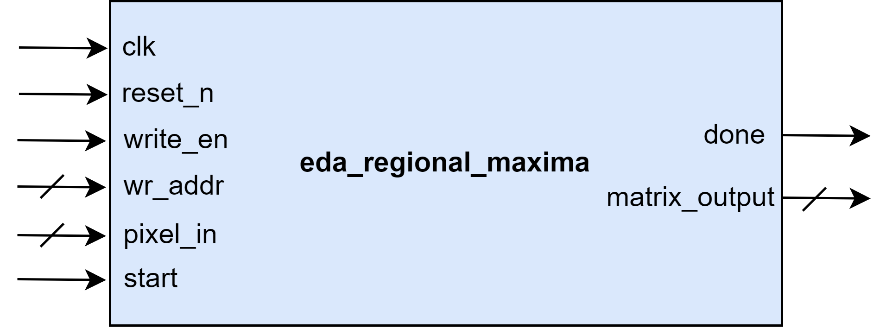


Figure 2.1 eda\_regional\_maxima block diagram

### 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 {i, j} |
| **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 |

### 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 values of **matrix\_output[i][j**] (i, j are positions of pixel) 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.

### Architecture

Figure 2.2 is **eda\_regional\_maxima**’s architecture.

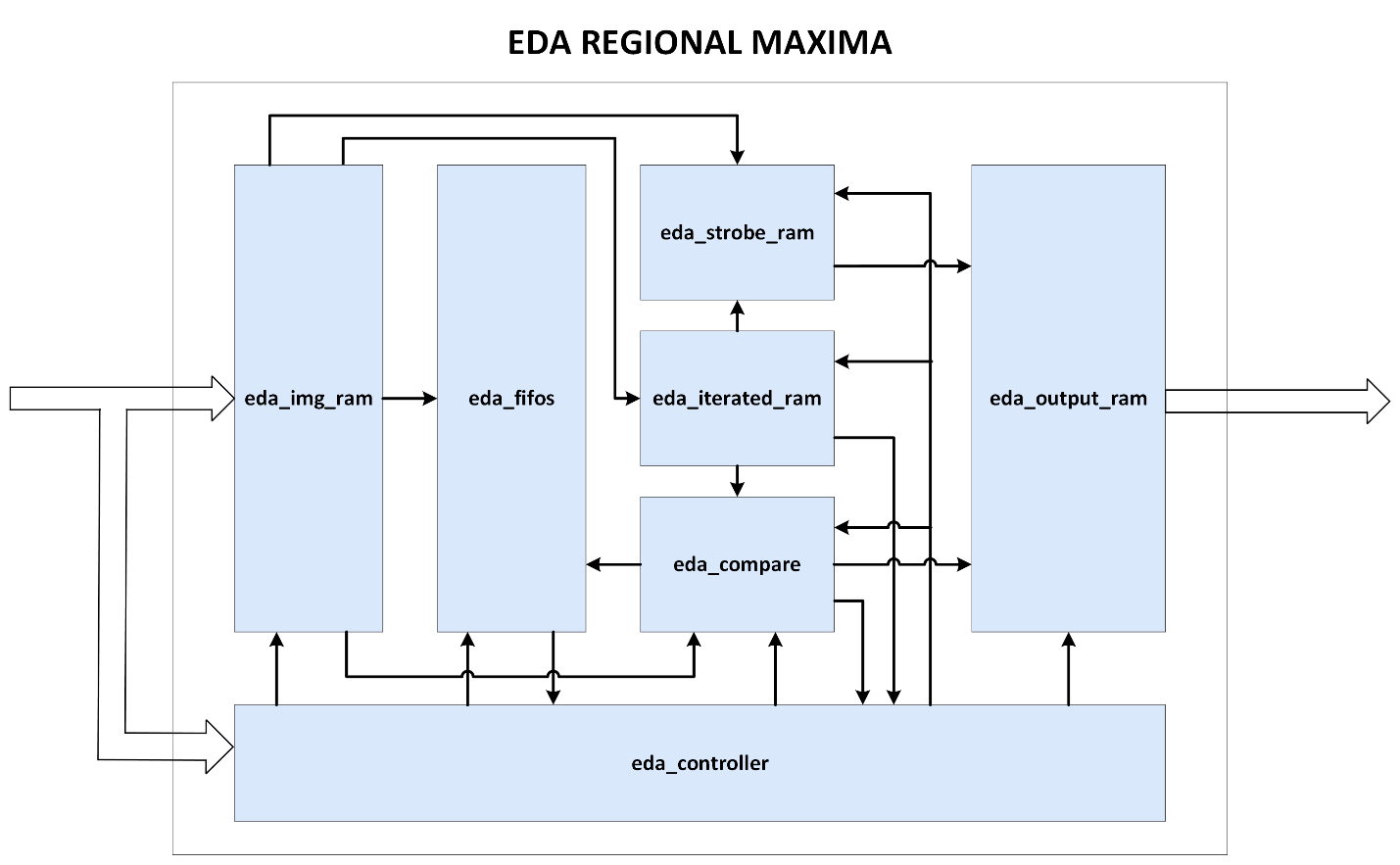


Figure 2.2 eda\_regional\_maxima architecture

**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.

### Timing waveforms

Figure 2.3 is the timing waveform of **eda\_regional\_maxima**.

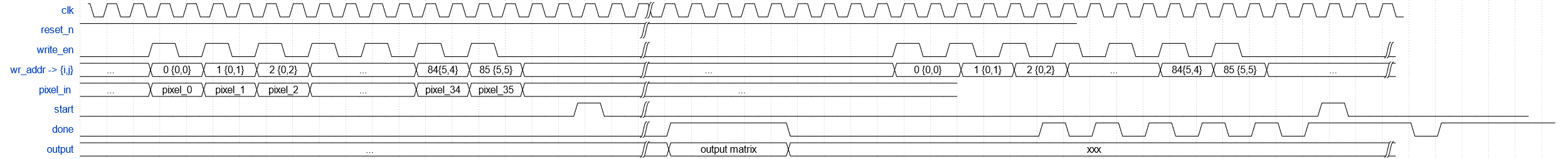


Figure 2.3 eda\_regional\_maxima timing waveform

## Sub module

### eda\_controller

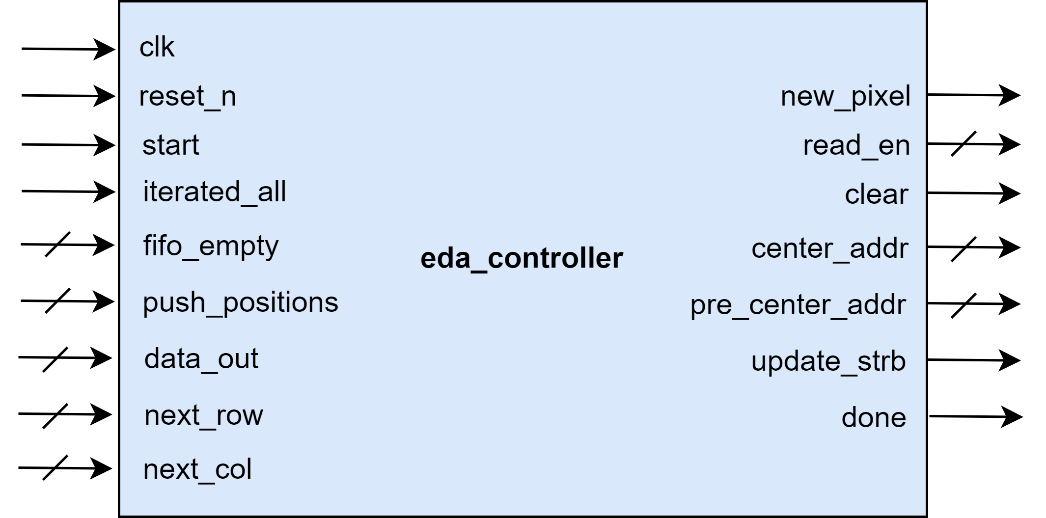


Figure 2.4 eda\_controller block diagram

#### 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 |

#### 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.

#### 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 |  |

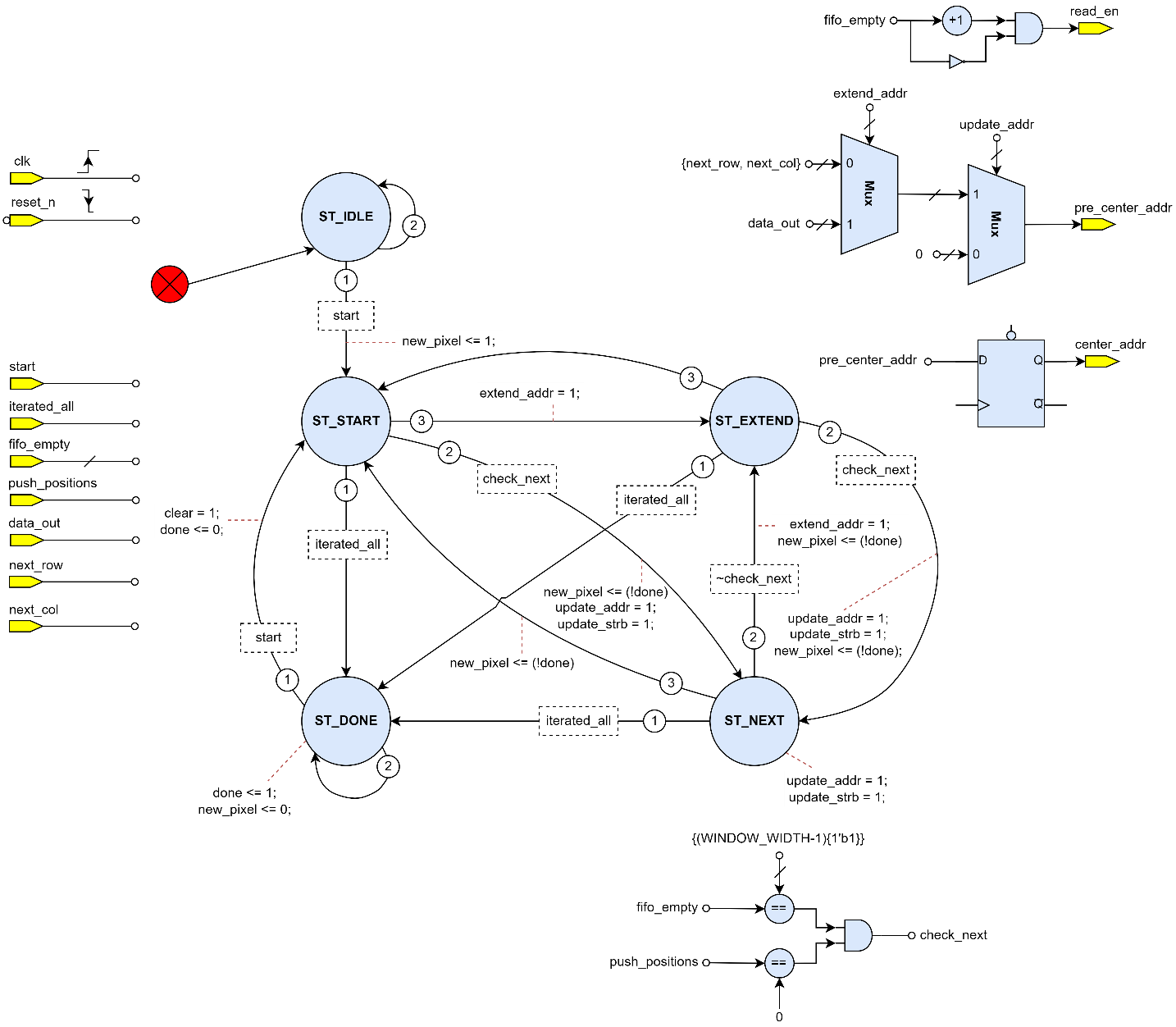


Figure 2.5 eda\_controller finite state machine

### eda\_compare

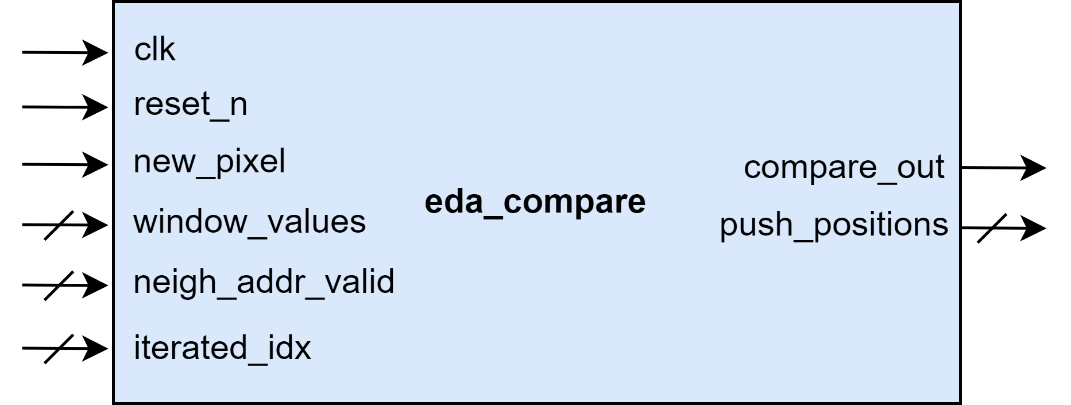


Figure 2.6 eda\_compare block diagram

#### 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 |

#### 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.

### eda\_fifos

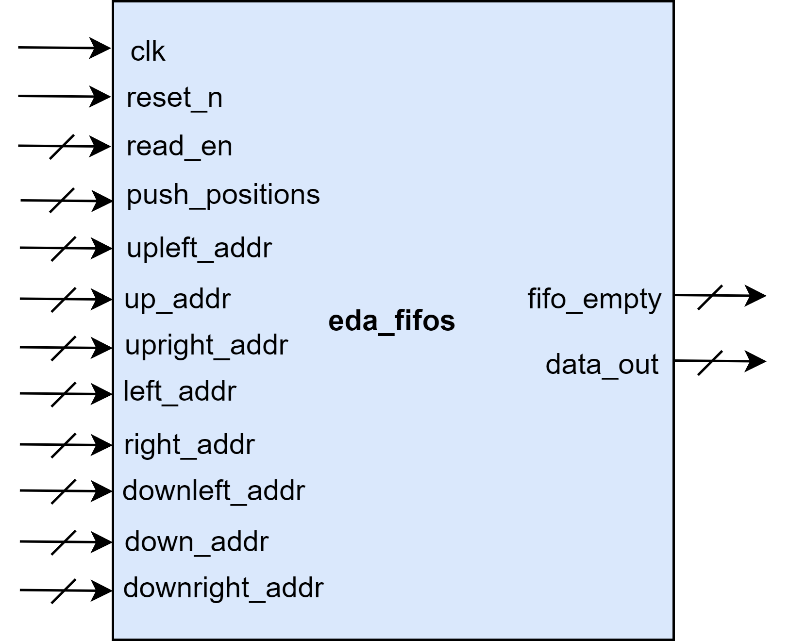


Figure 2.7 eda\_fifos block diagram

#### 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 |

#### 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.

### eda\_img\_ram

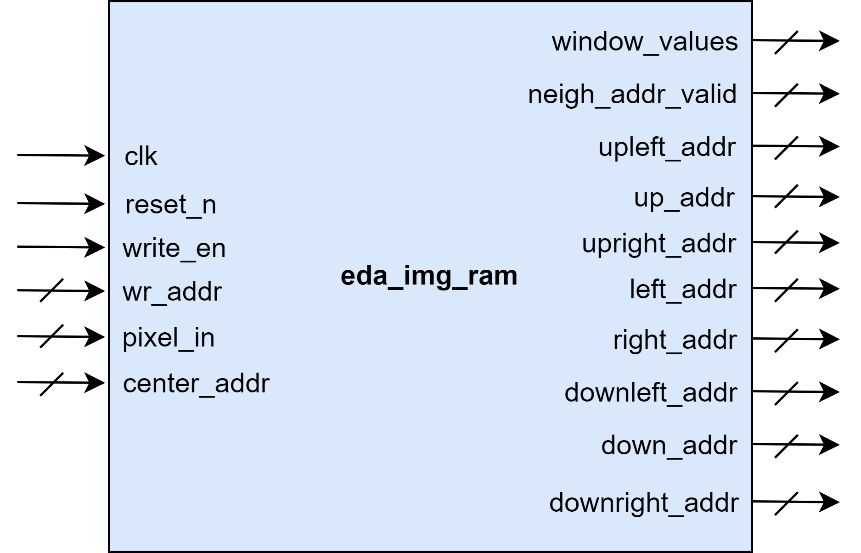


Figure 2.8 eda\_img\_ram block diagram

#### 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 |

#### 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**).

### eda\_iterated\_ram



Figure 2.9 eda\_iterated\_ram block diagram

#### 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 |

#### 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**.

### eda\_strobe\_ram



Figure 2.10 eda\_strobe\_ram block diagram

#### 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 |

#### 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.

### eda\_output\_ram

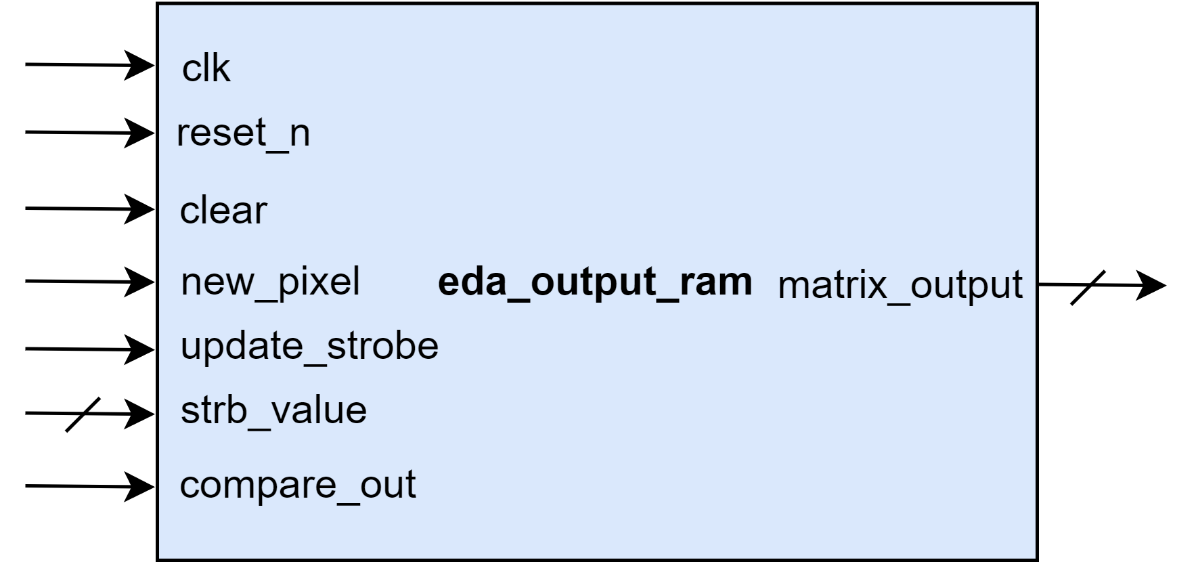


Figure 2.11 eda\_output\_ram block diagram

#### 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 |

#### Functional description

**eda\_output\_ram** updates the matrix output based on strobed signal.

# 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.

## Simulation result

### 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 1000000 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/**.

### Hardware

We create 1000000 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.**

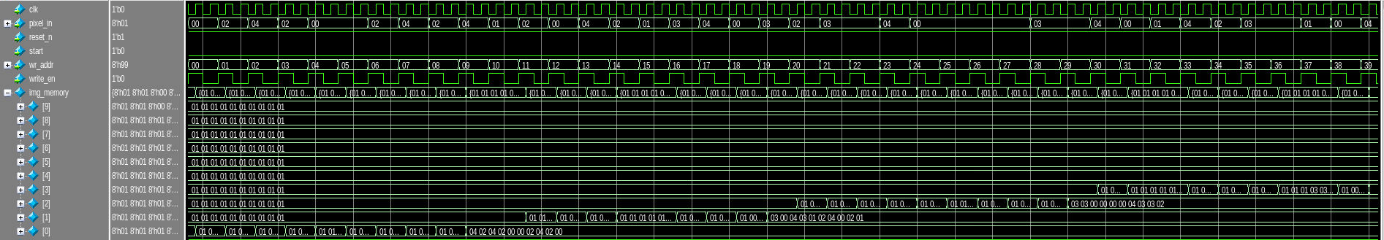


Figure 3.1 Load image into internal memory

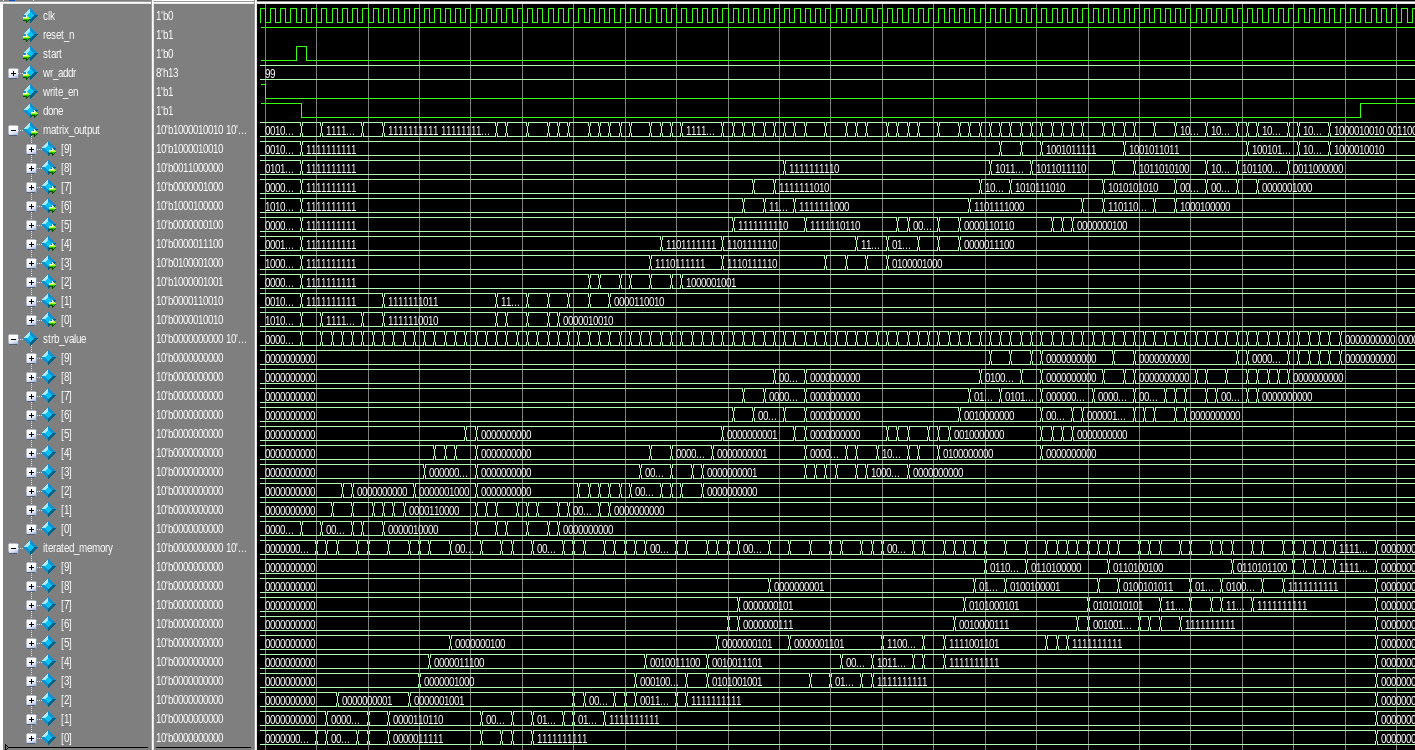


Figure 3.2 Waveform of 1 testcase

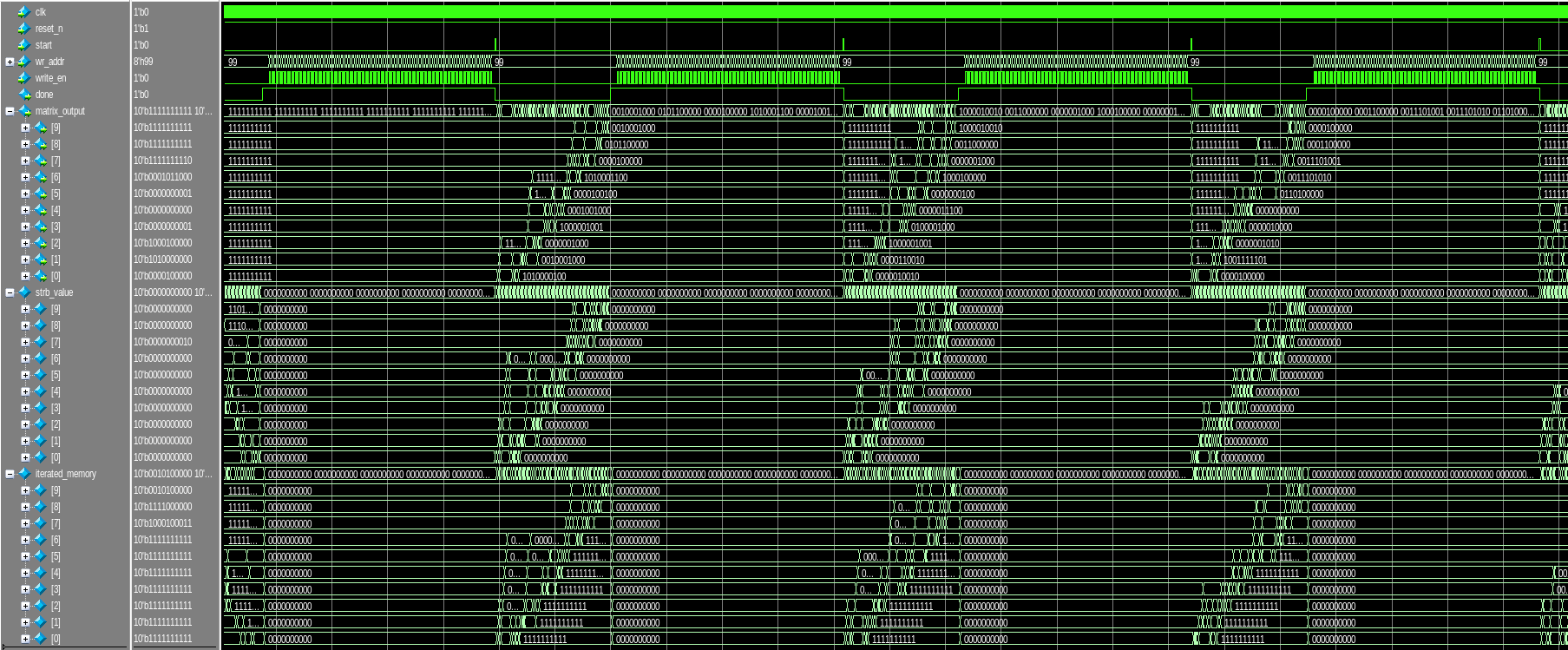


Figure 3.3 Waveform of multiple testcases

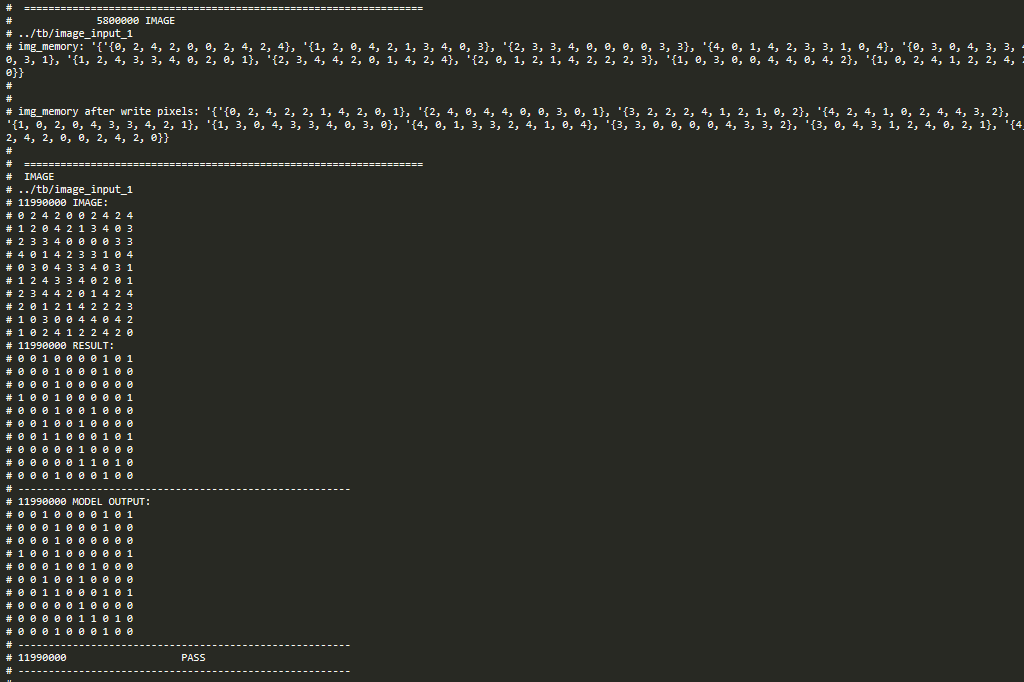


Figure 3.4 Result compared with model

## 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 |

# HDL CODE

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |