HARDWARE ARCHITECTURE FOR IMAGE CONVOLUTION FILTER WITH MULTIPLE PARALLEL KERNELS

B. FILIPESCU[1], D. POPESCU[1,2], NICOLETA ANGELESCU[2], I. TOMITA[1], R. DOBRESCU[1,2]
[1]Department of Applied Informatics University Politehnica Bucharest,
[2]Department of Electronics, Telecommunications and Energetics, Valahia University of Targoviste
E-mail: dan_popescu_2002@yahoo.com; nicoletaangelescu@yaho.com.

Abstract. ***The paper presents new digital hardware architecture for filtering images. The digital filters are implemented using 2 parallel convolution kernels. The system has been tested with an implementation of the Sobel filter.***

Keywords: hardware architecture, Sobel filter, convolution filter

1. INTRODUCTION

In some cases, in the images processing applications, images are taken from a camera, or are stored into an external memory which is large enough. In both cases, reading or downloading the images is done pixel by pixel, line by line.

Video cameras have the ability to provide pixels in a serial way, line by line. Most types of external memory are able to provide random access information but basically this kind of access is very slow (can be done at a very low frequency). There is a way to read information fast from an external memory, using BURST read mode. In this mode the memory provides information at a high frequency, but only from adjacent locations. This means that an external memory can provide pixels at a high frequency in the same way that a video camera does, i.e. pixel by pixel, line by line, if the image is stored in this way.

This paper is referring to the image filtering using 2D discrete convolution operation. If we consider an image "I" and a 3x3 convolution kernel "d", then the convolution operation for image "I" using kernel "d" is:

$$I_{filt}(x,y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} I(x+i,y+j) * d(1+i,1+j);$$

2. HARDWARE ARCHITECTURE OF THE CONVOLUTION OPERATIONS USING 2 PARALLEL KERNELS

The idea of this architecture is the ability to filter an image, using multiple parallel convolution kernels, pixel by pixel, while they are read, and to provide the results synchronously at the reading speed with certain latency (Figure 1).

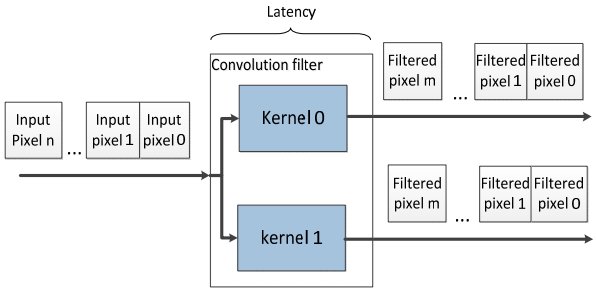

Figure 1. Image filtering using 2 parallel convolution kernels

In Figure 2 is presented a block scheme of a convolution filter architecture using 2 parallel convolution kernels.

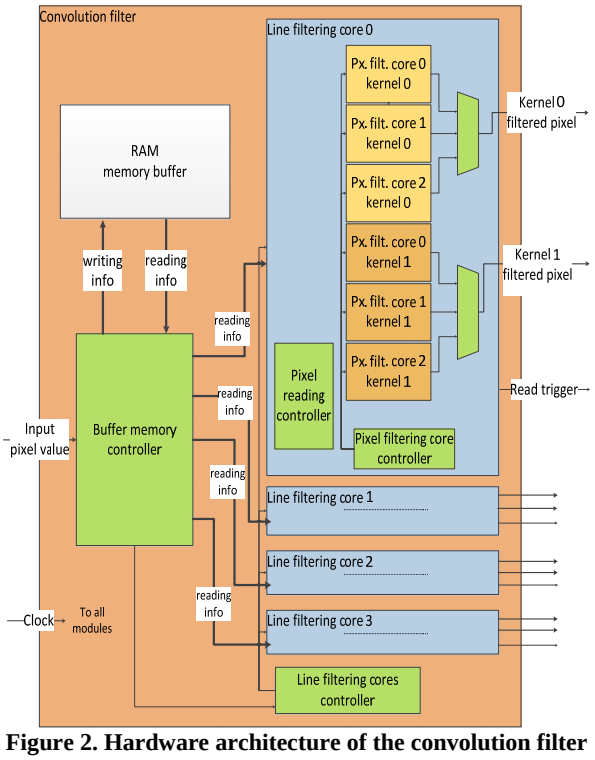

Figure 2. Hardware architecture of the convolution filter using 2 parallel kernels

The architecture components are:

- RAM memory buffer for storing input image's lines;
- RAM memory buffer controller, which has the purpose to provide controll signals for the memory buffer, write information and read information;

- 4 cores for image's line filtering, which are able to filter one input image line each;
- Line filtering cores controller, which has the purpose to make scheduling of the line filtering cores and set the input information for them;
- Every line filtering cores has:
  - 6 cores (3 for kernel 0 and other 3 for kernel 1) which have the purpose to filter one pixel;
  - a pixel filtering cores controller, which has the purpose to provide controll signals for the pixel filtering cores;
  - a pixel reading controller which has the purpose to provide the signals necessary for reading information from the RAM memory buffer (buffer line nr., pixel address from the current line e.t.c.);

Pixels' filtering is done by taking a 3x3 window (Figure 3) around every pixel from the input image and making the 2 convolution product.

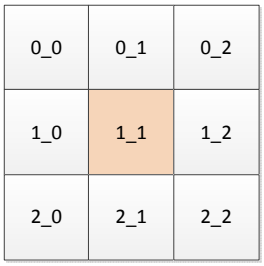

Figure 3. A 3x3 window around pixel 1_1

The algorithm for pixels filtering is implemented in a pipeline way and is presented in Figure 4. Every input pixel from the 3x3 window is processed in 3 clock cycles. On the first clock the input pixel is read and becomes available to be processed. On the second clock the current pixel is multiplied with the correspondent element from the convolution kernel (d (i,j) from Figure 4) and the result is stored into an auxiliary register (aux from Figure 4). On the third clock the result of the multiplication operation (stored in aux register) is accumulated into another register (filt_px from Figure 4).

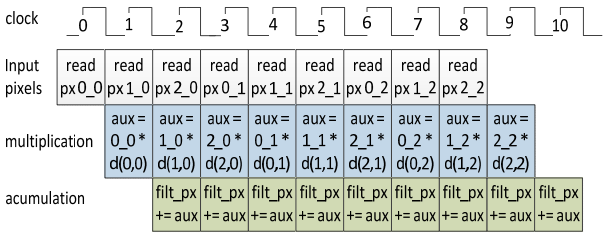

Figure 4. Block scheme for one pixel filtering algorithm

According to the figure 4 the clock numbers for filter one pixel is 11, i.e. 9 clock cycles for reading and 2 clock cycles latency.

The 3 pixel filtering cores, for each convolution kernel, are designed to work together for avoiding redundant reading of the information from the memory buffer to filter on line. In figure 5 are represented 4 lines with length of 7 pixels from an input image. Pixels 0_0, 1_0, 2_0 are necessary to filter pixel 1_1. Pixels 0_1, 1_1, 2_1 are necessary to filter pixel 1_1, but pixel 1_2, too. Pixels 0_2, 1_2, 2_2 are necessary to filter pixels 1_1, 1_2, but 1_3, too. According to those statements in some situations, the same input pixels are necessary to make 2 or 3 filtering operations. In order to eliminate redundant reading of the input pixels, are used 3 cores for one pixel filtering, and all pixels are read one time and used in parallel by all 3 pixel filtering cores. The synchronization algorithm of those 3 pixel filtering cores is presented in Figure 6.

Figure 5. Example of 3 image lines with size of 7 pixels

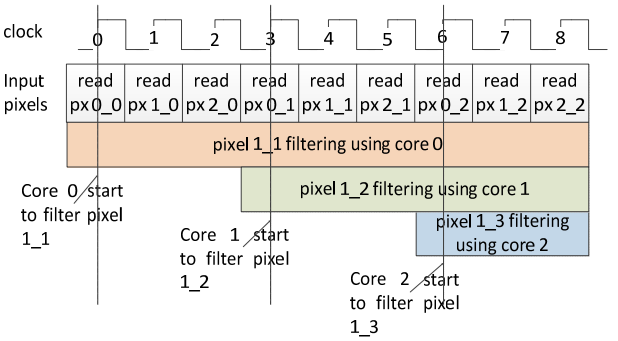

Figure 6: Block scheme for pixel filtering cores synchronization

The entire image filtering is done using the 4 line filtering cores, which work in parallel. To understand how those cores are synchronized, first, must be understood the structure of the memory buffer (Figure 8).

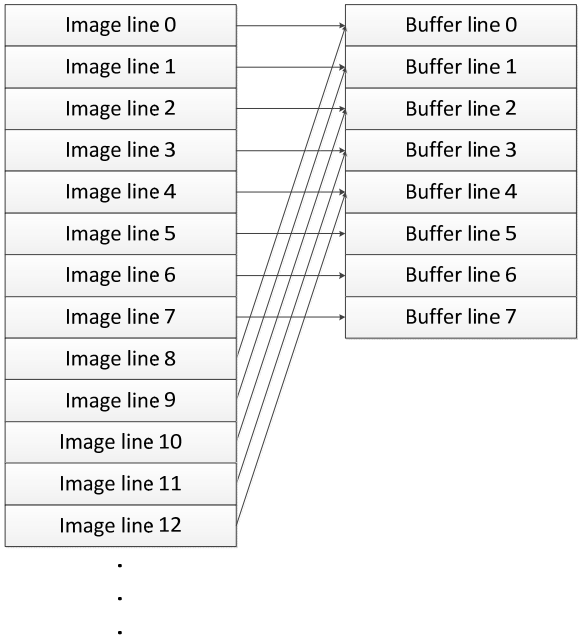

Figure 7. Input image lines mapping to the memory buffer lines

The memory buffer is composed of 8 lines in which are stored the lines of the input image, and has the purpose to provide random access to the image pixels at a high speed and to keep synchronized the processing and reading. The input image lines are mapped to the memory buffer lines using an algorithm similar with the direct mapping algorithm used at cache memory mapping in microprocessors [1] (Figure 7). Those 8 lines are actually 2 buffers consisting of 4 lines each. Input image processing and reading are 2 processes which are done in pipeline using the 2 memory buffers. The information written in lines 4, 5, 6, 7 is processed, while lines 0, 1, 2, 3 are written and vice versa. The number of 4 lines for memory buffer is necessary because line filtering process is taking a number of clock cycles a little more than 3*long thin pixels of a line image and fills half of the memory buffer that must be synchronized with a line filtering process. To maintain the robustness of the input image size, the number of lines from the buffer must be round to 4.

Every line from the memory buffer represents a standalone dual port memory, to avoid bottlenecks, and to give the possibility for every line filtering core to read information simultaneously, in 1 clock cycle, from different memory buffer lines. For example, when line filtering core 0 reads memory buffer line 0, line filtering core 1 will read memory buffer line 1, and so on.

Every line filtering core has 2 processing areas allocated in the memory buffer, from which reads information (Figure 8).

Line filtering core areas allocation is:

- Line filtering core 0: memory buffer lines 0, 1, 2 to filter line 1, and memory buffer lines 4, 5, 6 to filter line 5;

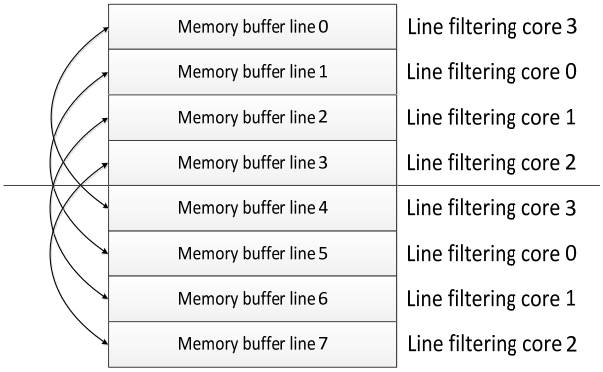

Figure 8. Memory buffer structure

- Line filtering core 1: memory buffer lines 1, 2, 3 to filter line 2, and memory buffer lines 5, 6, 7 to filter line 6;
- Line filtering core 2: memory buffer lines 2, 3, 4 to filter line 3, and memory buffer lines 6, 7, 0 to filter line 7;
- Line filtering core 3: memory buffer lines 3, 4, 5 to filter line 4, and memory buffer lines 7, 0, 1 to filter line 0;

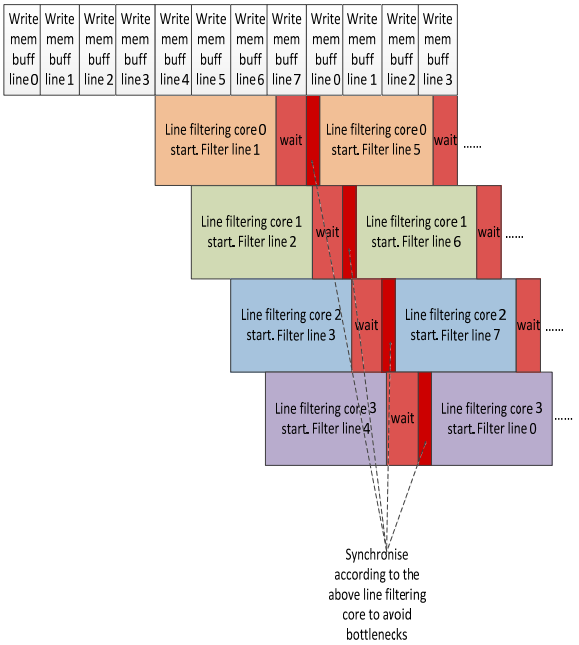

Figure 9. Block scheme for line filtering cores synchronization algorithm

In Figure 9 is presented the algorithm for line filtering cores synchronization. Initially, are filled the first 4 lines (lines 0, 1, 2, 3) of the memory buffer. While line 4 is filled, line filtering core 0 start to process the information from lines 0, 1, 2 to filter line 1. Line 1 is filtered in a clock cycles number approximately equal with (input image width * 3 + 2), so line 1 filtering

process will be finished while line 7 is filled. It is important to maintain a perfect synchronization between processing and memory buffer writing, to avoid the memory hazards, and for that, line filtering core 0 must wait until line 7 is filled completely, to start filtering from the line 5. The same algorithm is applied for line filtering cores 1, 2 and 3 according to Figure 9.

Until a line filtering core starts to filter a line, it must be synchronized according to the core from above, to avoid bottlenecks. There should be no situation where 2 line filtering cores read information from the same memory buffer line at the same time. Synchronization means that when core 0 reads information from the memory buffer line 0, core 1 must read information from the memory buffer line 1, core 2 must read information from the memory buffer line 2, core 3 must read information from the memory buffer line 3, and so on. Every line filtering core is synchronized according to the core from above: line filtering core 1 will be synchronized according to line filtering core 0, line filtering core 2 will be synchronized according to line filtering core 1, line filtering core 3 will be synchronized according to line filtering core 2, line filtering core 0 will be synchronized according to line filtering core 3, and so on. The latency of the algorithm is approximately (4*input image line width) clock cycles.

3. IMPLEMENTATION, SIMULATION AND TESTING

For the implementation were used Sobel filter convolution kernels for image gradient estimation.

Sobel kernels for OX and OY directions:

$$dx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ (for OX direction);}$$

$$dy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \text{ (for OY direction);}$$

The implementation of the convolution filter hardware architecture using Sobel kernels was done using Verilog HDL [2] and it was simulated using ModelSim [5], [6]. For the validation of the functionality it was used an automatic test bench environment, implemented also in Verilog [3], [4]. The test environment is presented in Figure 10.

Test bench components are:

- Sobel filtering core (DUT)
  - Sobel filtering core module instantiation, DUT (Design Under Test);
- Matrix generator
  - Generates random matrices 8 bits length values, for matrices with configurable size;

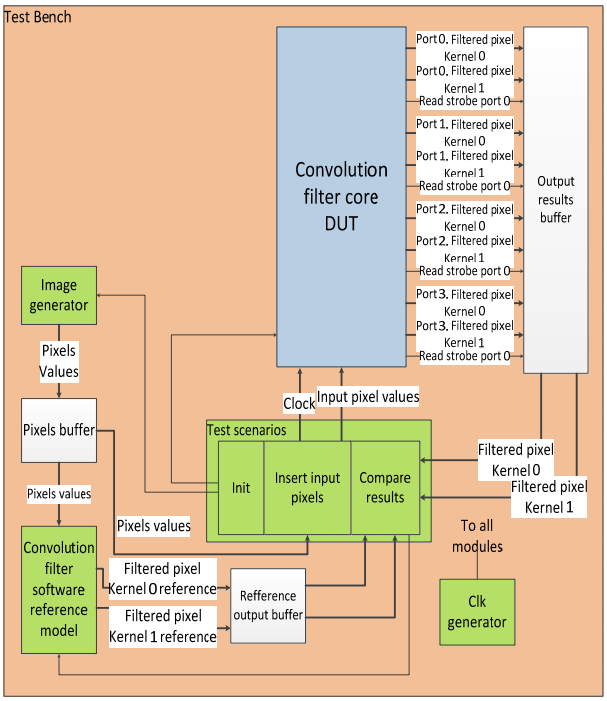

Figure 10. Block scheme for the automatic test bench environment

- Matrix elements buffer:
  - storage environment for generated matrix elements;
- Sobel filter software reference model:
  - Software implementation of the Sobel filtering algorithm;
  - It is used to generate the reference results for sobel filtering algorithm;
- Output results buffer:
  - storage environment for the Sobel core DUT results, results which should be tested;
- Output reference results buffer:
  - storage environment for the refference results;
- Test scenario (Figure 11):
  - controls the input data for Sobel filtering module;
  - controls the Sobel filter software reference model to generate benchmark result;
  - compares benchmark results with the Sobel core DUT results and displays error and timing measurements information;

Init:

- starts clock generator;
- initializes all signals;
- generates a random matrix with a specified size, and elements on 8 bits;

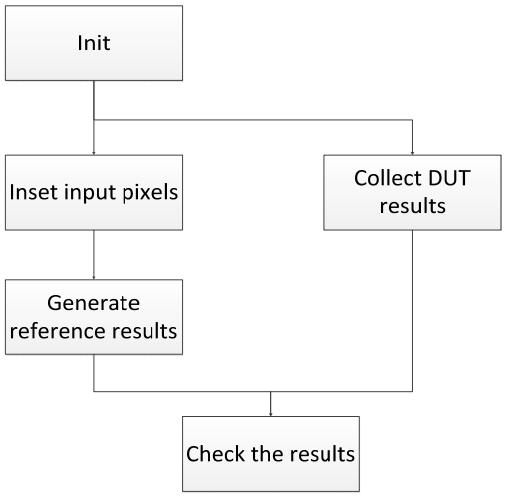

Figure 11. Test scenario

Inset input pixels:

- sends to the convolution core the elements from the generated matrix, pixel by pixel, line by line;
- calculates latency;

Collect DUT results:

- Collects the results from the convolution core DUT and store them into the output results buffer;

Generate reference results:

- Starts convolution core algorithm software reference to generate the benchmark results;
- Stores the results into the output reference results buffer;

Check the results:

- Compares the convolution core DUT results with the benchmark results;
- Displays the error messages;
- Displays performance results:
  - clock cycles numbers;
  - measured latency;

4. RESULTS

In figure 12 are presented the results of one simulation using ModelSim. The input image was generated random and with size of 320x240 pixels. The total number of image pixels is 76800 and according to Figure 12 are necessary 78093 clock cycles to filter the entire image. So, the latency is 1293 clock cycles, i.e. approximately (4*input image line width).

5. REFERENCES

[1] David A. Patterson, John E. Hennessy, “The basic of caches,” in “Computer organization design”, 3th edition, University of California Berkeley, 2005.

[2] Samir Palnitkar, “Verilog HDL A guide to Digital Design and Synthesys”, SunSoft Press 1996.

[3] Peter J. Ashenden, “Digital Design An Embedded Systems Approach Using Verilog”, Elsevier 2008.

[4] Alexander Miczo, “Digital Logic Testing and Simulation”, 2th edition, Wiley 2003.

[5] Altera “Using ModelSim to Simulate Logic Circuits for Altera FPGA Devices”, Altera corporation 2011.

[6] Altera “Introduction to Simulation of Verilog Designs Using ModelSim Graphical Waveform Editor”, Altera corporation 2010.

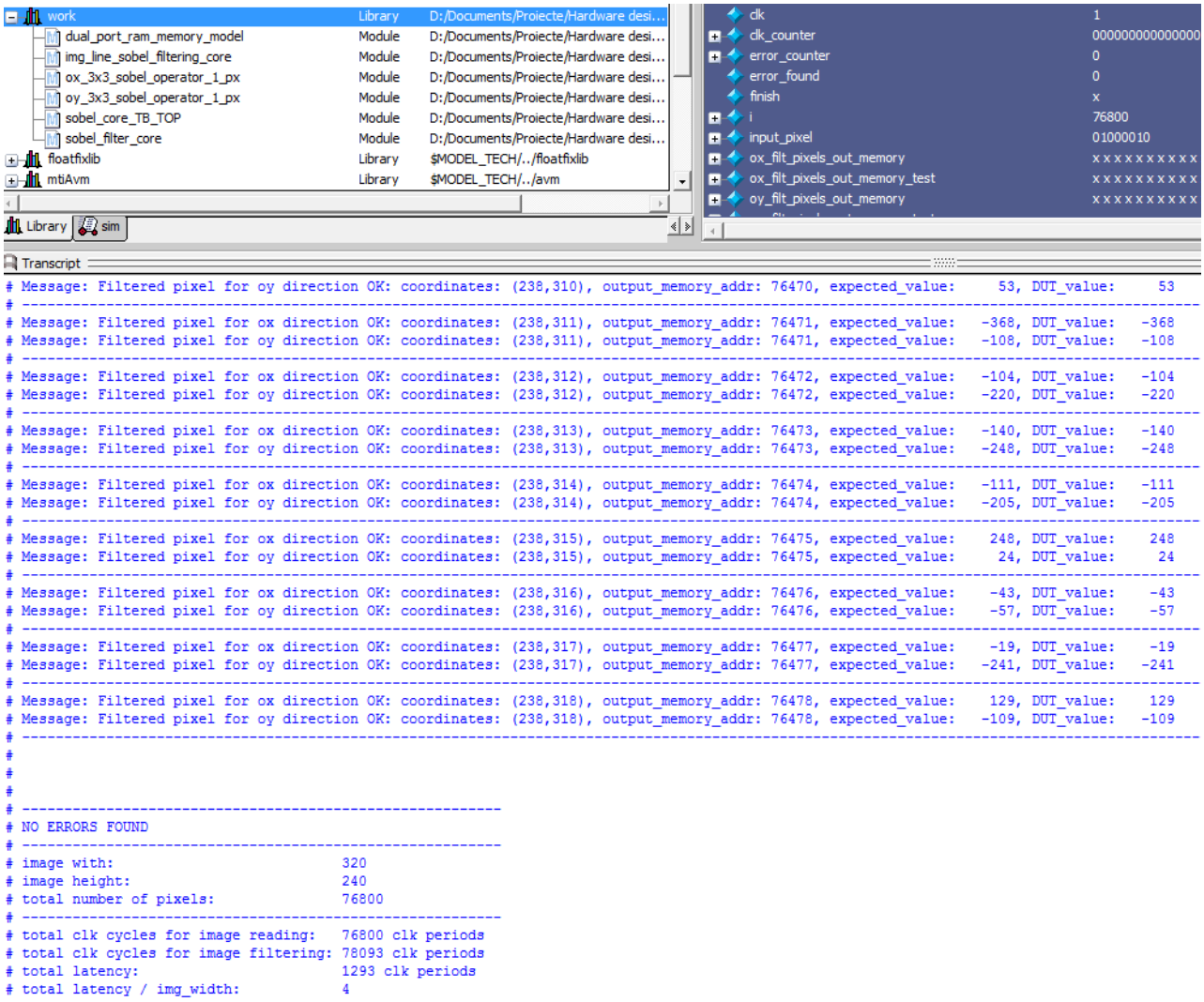

Figure 12. Simulation results