2010

FPGA BASED PARALLEL IMPLEMENTATION OF STACKED ERROR DIFFUSION ALGORITHM

Rishvanth Kora Venugopal
University of Kentucky, rkora3@uky.edu

ABSTRACT OF THESIS

FPGA BASED PARALLEL ARCHITECTURE IMPLEMENTATION OF STACKED ERROR DIFFUSION ALGORITHM

Digital halftoning is a crucial technique used in digital printers to convert a continuous-tone image into a pattern of black and white dots. Halftoning is used since printers have a limited availability of inks and cannot reproduce all the color intensities in a continuous image. Error Diffusion is an algorithm in halftoning that iteratively quantizes pixels in a neighborhood dependent fashion. This thesis focuses on the development and design of a parallel scalable hardware architecture for high performance implementation of a high quality Stacked Error Diffusion algorithm. The algorithm is described in 'C' and requires a significant processing time when implemented on a conventional CPU. Thus, a new hardware processor architecture is developed to implement the algorithm and is implemented to and tested on a Xilinx Virtex 5 FPGA chip. There is an extraordinary decrease in the run time of the algorithm when run on the newly proposed parallel architecture implemented to FPGA technology compared to execution on a single CPU. The new parallel architecture is described using the Verilog Hardware Description Language. Post-synthesis and post-implementation, performance based Hardware Description Language (HDL), simulation validation of the new parallel architecture is achieved via use of the ModelSim CAD simulation tool.

KEYWORDS: Halftoning, Stacked Error Diffusion, Verilog, Parallel Architecture, HDL Simulation Validation.

RISHVANTH KORA VENUGOPAL

12/02/2010

FPGA BASED PARALLEL ARCHITECTURE IMPLEMENTATION OF STACKED ERROR DIFFUSION ALGORITHM

By

RISHVANTH KORA VENUGOPAL

Dr. J. Robert Heath
Director of Thesis

Dr. Daniel Lau
Co-Director of Thesis

Dr. Stephen Gedney
Director of Graduate Studies

12/02/2010
Date

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date

THESIS

Rishvanth Kora Venugopal

The Graduate School

University of Kentucky

2010

FPGA BASED PARALLEL IMPLEMENTATION OF STACKED ERROR DIFFUSION ALGORITHM

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Electrical Engineering in the
College of Engineering at the University of Kentucky

By

Rishvanth Kora Venugopal

Lexington, Kentucky

Director: Dr. J. Robert Heath, Associate Professor of Electrical Engineering

Co-Director: Dr. Daniel Lau, Associate Professor of Electrical Engineering

Lexington, Kentucky

MASTERS THESIS RELEASE

I authorize the University of Kentucky
Libraries to reproduce this thesis in
whole or in part for purposes of research.

Signed: _____

Date: _____

Dedicated to my Family, Teachers and Friends.

ACKNOWLEDGMENTS

I express my sincere gratitude to my advisors Dr. Robert Heath and Dr. Daniel Lau, whose encouragement, guidance and support throughout my research work helped me develop a through understanding of the subject.

I would like to thank Dr. Meikang Qui for serving on my thesis committee. I would also like to thank my colleagues at Computer Architecture Laboratory who supported me during technical difficulties.

I sincerely thank my family members and friends who showed me kindness and generosity throughout my life.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

A digital image consists of millions of colors combined to form a continuous tone image. In order to print an image using a printer where the device has a limited number of colors (inks) available to represent the original image, a technique known as halftoning was invented to convert the original image to a simple binary format. The Error Diffusion technique and Serpentine scanning methodology used in halftoning makes parallel processing implementation cumbersome as it creates huge inter-pixel dependencies along with the generation of errors at each pixel location that has to be stored in the memory for subsequent processing. This thesis focuses on the development and design of a high speed parallel hardware architecture that implements a proprietary high quality halftoning 'C' algorithm in FPGA technology. The purpose of this study is to improve the execution speed of the algorithm by executing it on an FPGA as opposed to a conventional CPU that takes enormous processing time to execute.

1.1 Background

There have been many improvements in halftoning algorithms to achieve the best quality without compromising the performance of the hardware devices running the algorithms. Performance of the device depends entirely on the type of algorithm and the scanning method used. This section of the chapter gives a detailed explanation of the algorithm used and ways to improve the processing performance of the implementing hardware system.

1.1.1 Halftoning

Halftoning is a technique used to convert a continuous-tone image into a series of black and white dots. Image reproduction devices such as printers and monitors are constrained to a few colors and cannot print a digital image that consists of millions of colors. Thus, halftoning transforms the original image into a binary image containing only 1’s and 0’s where a '1' at a particular pixel suggests a black dot to be printed and 0 means that the corresponding pixel should be empty. In the case of color image reproduction, the

halftoning is performed on each of the color channels, namely Red, Green and Blue (RGB) or Cyan, Magenta, Yellow and Key (CMYK). Thus, '1' in a color halftoned image suggests a particular channel to be printed. There are many methods [1] in which halftoning is performed on images and some of them include AM & FM Halftoning, Table Halftoning, Threshold Halftoning, Ordered Dithering, Error Diffusion, Iterative Halftoning, Hybrid AM-FM Halftoning and Multilevel Halftoning. This thesis deals with the halftoning algorithm that uses the basic Error Diffusion technique [2].

1.1.2 **Error Diffusion**

A common method for producing halftoned images is the Error Diffusion technique invented by Floyd and Steinberg [2] where the error from each pixel is dispersed to the neighboring pixels. The output value of each pixel depends on the input pixel and the diffused error value from the previous pixel. Figure 1.1 shows the error diffusion method where *c* and *h* represent the continuous and the halftoned images respectively. The input pixel *c* is added with the Previous Pixel Error value *p* and compared with a threshold. The output *h* is obtained from the comparison and the Error *e* is calculated by subtracting the current output value *h* from the combined value of *p* and *c*. Further, the Error *e* is multiplied with the weight filter and diffused across the neighboring pixels. The Error dispersion and the Error Weight Filter is shown in Figure 1.2.

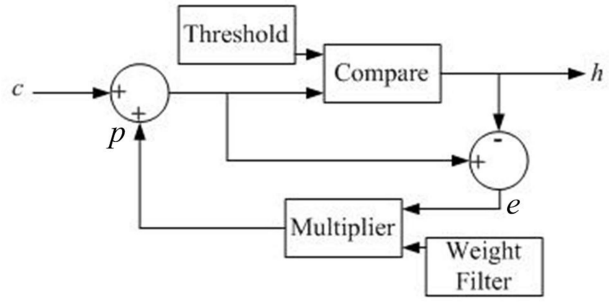

Figure 1.1: Floyd-Steinberg Error Diffusion

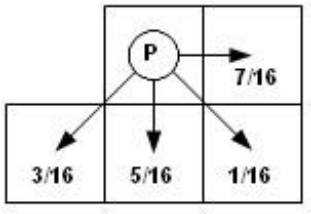

Figure 1.2: Error Filter

1.1.3 **Image Scanning Techniques**

The two main techniques used in scanning an image are the **Line Raster** (left-to-right, top-to-bottom approach) and the **Serpentine Raster** (left-to-right, right-to-left) techniques. A **Line Raster** is the process of reading an image starting left and ending right for each row from top till bottom of the image. This type of scan results in an output halftone consisting of checkerboard patterns, worms and other geometric artifacts. **Serpentine Scan** is the process of scanning even rows of an image in left-to-right fashion and odd rows in right-to-left fashion. This research deals with a serpentine scan methodology because this technique results in fewer artifacts. The downsides of using this technique are that it makes parallel processing even more burdensome and the memory required to store the errors generated at each pixel location is large. Figure 1.3 shows the line raster scan process where P_0 till P_{10} are the pixels in the first row and P_{11} till P_{21} are the pixels in the second row.

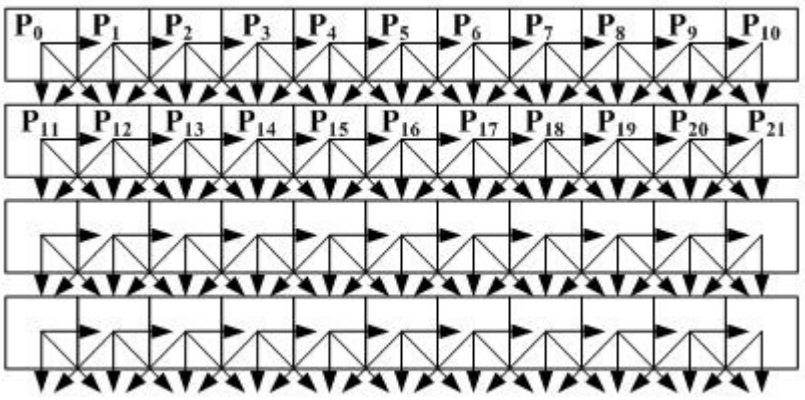

Figure 1.3: Line Raster

When a line raster is implemented on the image, the processing starts from P_0 and reaches P_{10} and again from P_{11} and reaches P_{21}. This type of scan has a high probability

of parallelism in which the pixel P_{11} can be processed right after the pixels P_0 and P_1 are processed as the pixel P_{11} depends on P_0 and P_1 alone. Thus the inter-pixel dependency in this method is kept to a minimum. Figure 1.4 shows the serpentine raster technique where pixel P_{11} cannot be processed until all the pixels from P_0 till P_{10} and P_{21} till P_{12} are processed. As a result, the errors obtained at each pixel location must be stored in memory till the specified pixel is processed. Thereby, this technique requires a large memory space which in turn depends upon the image size being processed.

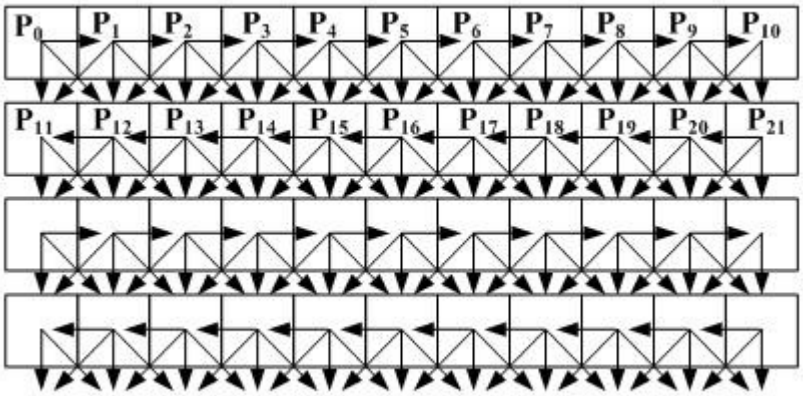

Figure 1.4: Serpentine Raster

1.1.4 **Blue-Noise**

Blue-noise is any noise with the least low frequency element and absolutely no intense spikes of energy. Ulichney [3] studied the spectral charateristics and noted their predominantly high frequency content, a characteristic he called Blue-Noise. This makes it an important noise in halftoning as the retinal cells in the human eye are organized in a manner similar to the blue-noise which results in great optical interpretation. The arrangement of the droplets in a halftoned image creates an optical illusion which the human eye mistakes for a continuous tone image. The introduction of blue-noise in error diffusion has a great impact on the quality of the halftoned output image. It makes the resulting image appear visually smooth.

1.1.5 **Blue-Noise Halftoning**

Many algorithms have been implemented to produce halftone patterns with blue-noise attributes. Blue-noise halftoning/dithering constitute an array of minority pixels that are

uniformly distributed that results in halftones that lacks regularity and low frequency elements. In the error diffusion algorithm proposed by Floyd and Steinberg [2], a quantizer is used that compares the input pixel value with a threshold to determine the value of the corresponding pixels. The quantizer error is calculated by subtracting the input pixel with the threshold value and is diffused to the neighboring pixels using an error filter $P = [(7/16), (1/16), (5/16), (3/16)]$ shown in Figure 1.2. This process is executed on all the pixels till the complete image has been processed. The output of the algorithm applied to a gray-scale ramp is shown in Figure 1.5. Figure 1.5 shows that when the image is scanned using Line Raster method, the final halftone consisted of checkerboard patterns, worms and other geometric artifacts. Thus to avoid visual artifacts arising from the conventional approach, a serpentine scanning approach is implemented and the threshold error diffusion is altered depending on the outputs of the previously processed pixels. The threshold in error diffusion technique can be altered depending upon the previous outputs [5] or by the intensity of the present pixel as indicated by Eschbach and Knox [6]. Eschbach [7] and Ostromoukhov [8] proposed changes in shape of the filter and weights dependent on the inputs. Li and Allebach [9] proposed a technique where the thresholds and weights are optimized based on the model for the human visual system. The current algorithm under discussion uses a design proposed by Ulichney [10] where a serpentine scan and randomness (R_1, R_2) in weights of the error filter are introduced. The weights are calculated as $[P_1 + R_1, P_2 - R_2, P_3 - R_1, P_4 + R_2]$, where $R_1 = (5/16)\ U[-1,1]$, $R_2 = (1/16)\ U[-1,1]$ and $U(m,n)$ represents a uniformly distributed random variable in the interval $[m,n]$. This randomness in the error filter eliminates most of the geometric and checkerboard artifacts in the resultant output image. The output of the algorithm applied to a gray-scale ramp is shown in Figure 1.6. The original blue-noise model is implemented in Floyd-Steinberg's technique, whereas the technique implemented by Ulichney is a realization of the model proposed by Lau and Ulichney [11].

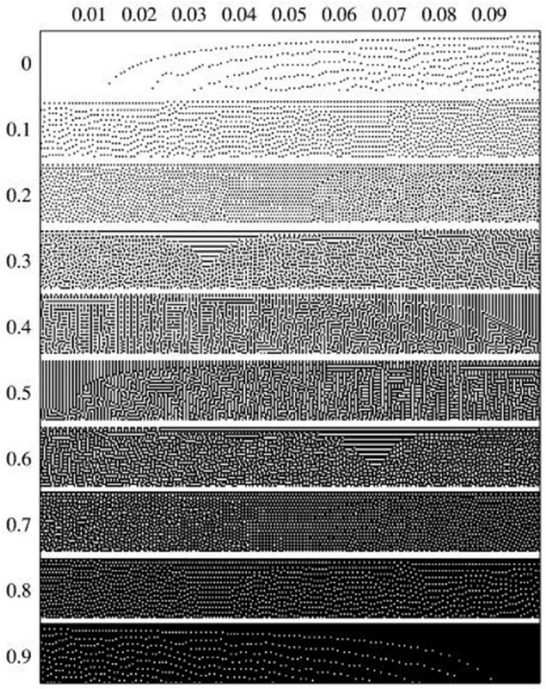

Figure 1.5: Halftone of Gray-Scale ramp generated with Floyd-Steinberg Error Diffusion. Adapted from [4]

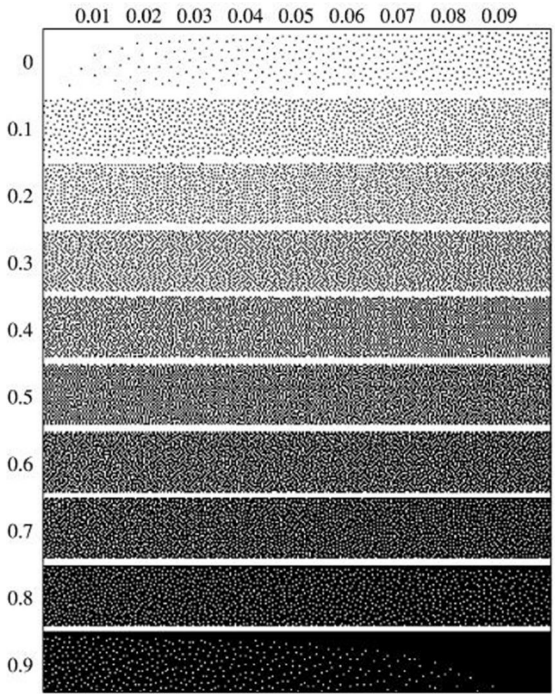

Figure 1.6: Halftone of Gray Scale Ramp generated with Ulichney's Error Diffusion. Adapted from [4]

1.1.6 **Multitoning**

Multitoning is the process of reproducing an image using multiple inks [4]. A multitone is an embedded array of halftone patterns with different inks printed on top of each other which is similar to color halftoning where 3 or more primary halftones are superimposed in order to achieve the illusion of a continuous tone color image. A multitone dither pattern with N different inks of intensities $(g_1, g_2,.....,g_N)$ where g is the gray level, arranged starting from lightest (white, nothing printed having intensity $g_1 = 0$) to the darkest (black, printed pixel having intensity $g_N = 1$) consists pixels of $N+1$ different intensities. The main disadvantage in superimposing halftones is the emergence of a low-frequency noise called Moiré. This anomaly occurs in dispersed dot patterns as an irregularity in the arrangement of pixels which is referred to as **S*tochastic moiré***. The irregularity is caused due to the difference in placement of dots in the superimposed

halftones. Wang and Parker [29] suggested that the combination of two blue-noise patterns doesn't inherently produce a good quality pattern, but depends on both the spectrum of individual patterns and the interrelationship between them. According to threshold decomposition, a discrete signal which accepts one of *k* possible values can be expressed as the weighted sum of *k - 1* binary signals. Consider multitoning where *M* is the multitone dither pattern and the array of halftones $H_i|_{i=1}^{N}$ is defined as

$$H_i[n]=\begin{cases}1, & if\ M[n]\geqslant g_i\\ 0, & else\end{cases} \qquad (1.1)$$

The halftone H_i describes the threshold decomposition of the multitone *M* at level *i*. Equation 1.1 states that a printed pixel in H_i implies that a printed pixel of intensity g_i or darker occurs in the multitone in the same location and also means that there is a printed pixel in the same location in H_j for all $j \leq i$. Thus the decomposition of the multitones into an array of halftones is done by satisfying the stacking constraint or in other words the halftones are constrained to a stack. It can be said that the multitone is a linear unification of stacked blue-noise patterns. The multitone can be described in terms of its threshold decomposition representation as

$$M[n]=\sum_{i=1}^{N} d_i H_i[n] \qquad (1.2)$$

where $d_i = g_i - g_{i-1}|_{i=1}^{N}$ are the relative differences between intensities of the printable inks. Figure 1.7 shows the decomposition where the multitone *M* is a 3×3 image printed with three inks with intensities $(g_1, g_2, g_3) = ((1/3),(2/3),1)$.

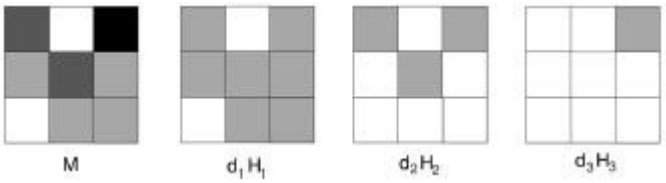

Figure 1.7: Decomposition of 3-ink multitone M in a series of Halftones satisfying the stacking constraint.

Adapted from [4]

1.1.7 **Blue-Noise Multitoning with Stacked Error Diffusion**

Many halftoning developments have been proposed as extensions to previously developed halftonig algorithms. Some of them are ; modification of the error diffusion algorithm by replacing the binary thresholding by a multilevel quantizer [12], correlated error diffusion applied to channels that represents available inks (Faheem [13]), screening (***Screening*** is the process of representing lighter degree of color as a tiny dot of ink) applied in multitoning using Bayer dither arrays [12] and clustered-dot dithering [14]. The concept of gray level distribution is introduced in some of the algorithms where the amount of each of the printable inks used to generate a certain gray level is defined and controlled accordingly. This thesis research algorithm uses a concept of gray level distribution where the amount of printable inks (colors) used to create a certain gray level is known before hand. For a multitone to be visually pleasant and optimal, the dots of different inks should be positioned in a correlated pattern. A technique similar to threshold decomposition is used to divide multitones into halftones and to synthesize them to make sure that the resultant picture is flawless. Assuming a constant block of intensity *g* to be reproduced using the inks $g_i|_{i=1}^{N}$ in segments $p_i|_{i=1}^{N}$, the intensity of the block is represented as

$$g = \sum_{i=1}^{N} g_i p_i(g) = \sum_{i=1}^{N} d_i \mu_i(g) \qquad (1.3)$$

where $\mu_i(g) = \sum_{j=i}^{N} p_j(g)$ and $d_i = g_i - g_{i-1}$ .Consider a block of intensity $\mu_1(g)$ halftoned using blue noise, the output dither pattern will have the same characteristics as H_1. The same process is executed for a block with intensity $\mu_2(g)$ with the condition that the resulting halftone should *stack* (depend) on the first halftone. Thus the output dither pattern has the same characteristics required by H_2. This depends on the number of levels in the given image and assuming there are *i* levels, the same procedure is repeated for the remaining $\mu_i(g)|_{i=3}^{N}$ where the *i*th halftone stacks on the *i* – *1*st halftone and the result will be a series of *N* halftones. The linear unification of all the *i* halftones gives the best blue-noise multitone. The method to multitone a continuous tone image *Y* is shown in

Figure 1.8. Firstly, the printing method provides the ink intensities g_i and the corresponding concentrations $p_i(g)$ are to be determined before hand by the user. Secondly, the gray levels are mapped to corresponding droplet densities. This can be accomplished with the help of a look-up-table.

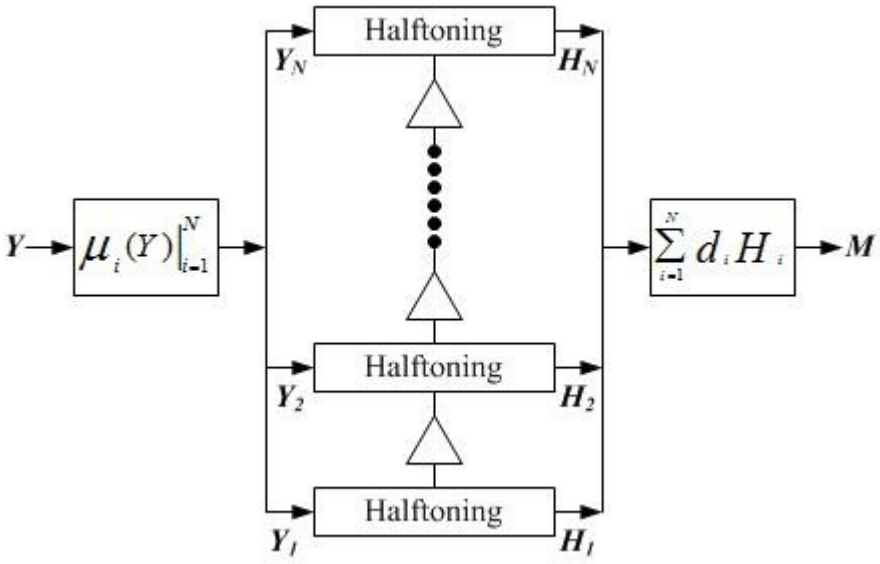

Figure 1.8: A Continuous tone image Y divided into N components resulting in a final halftone M [4]

Thirdly, halftoning is carried out with a suitable algorithm which in this thesis is ***Blue-Noise Multitoning with Error Diffusion*** by taking the stacking constraint into account. For example, to get H_2, Y_2 needs to be halftoned which stacks on the halftone H_1. The same procedure is followed with the remaining $Y_i|_{i=3}^{N}$ continuous-tone images. Finally, Equation 1.2 is used to obtain the final multitone. The algorithm in this thesis research work uses blue-noise multitoning with error diffusion and in order to generate the multitones by this method, the stacking constraint should be incorporated in the pixel quantization satisfying Equation 1.4 below.

$$H_i[n]=\begin{cases}1, & if \quad Y_i[n] + H_i^p \geq \frac{1}{2} \text{ and } H_{i-1}=1 \\ 0, & else\end{cases} \qquad (1.4)$$

where $H_i^p[n]$ is the error diffused to the pixel $H_i[n]$ and $i = 1, \ldots, N$. If $i=1$, it is assumed that $H_0[n]=1 \; \forall \, n$. Figure 1.9 shows the implementation of the stacked error diffusion algorithm where the continuous input image *c* is added to the corresponding diffused Error, the result is compared with the Threshold and the Previous Level Output thus producing a halftone *h* based on the stacking constraint. The Error *e* in the input pixel *c* and the output *h* is calculated. Hence the Error produced *e* is multiplied with the Perturbed Weight Filter resulting in a diffused Error dispersed to soon-to-be-processed pixels. The Perturbed Weight Filter eliminates the artifacts that arise in a normal error diffusion method.

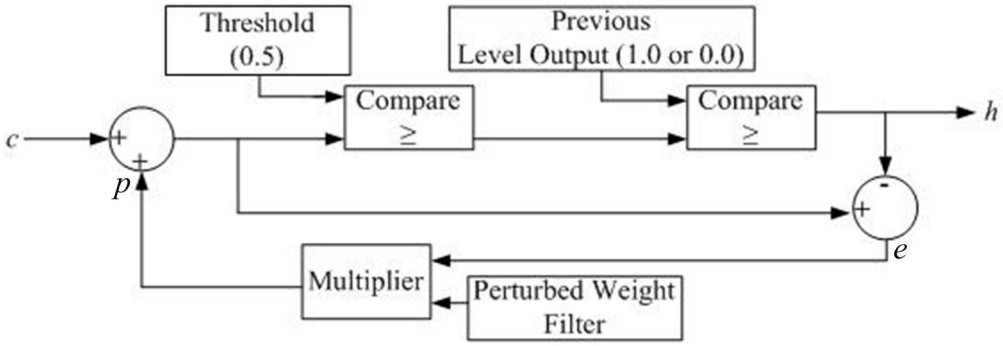

Figure 1.9: Stacked Error Diffusion

1.2 **Previous Research on FPGA Implementation of Halftoning Algorithms**

There have been a number of proposals for implementing halftoning algorithms on Field Programmable Gate Array (FPGA) technology. Metaxas [15] proposed an optimal Error-Diffusion parallel algorithm for Digital Halftoning implemented in MasPar data-parallel computers (SIMD – Single Instruction Multiple Data). Yuefeng Zhang [16] presented a parallel Error-Diffusion algorithm, known as Line Diffusion implemented using a massively parallel algorithm. Jae-woo Ahn and Wonyong Sung [17] proposed a multimedia processor based implementation of an Error-Diffusion Halftoning Algorithm where multiple pixels are processed simultaneously using subword-parallel arithmetic and logic unit architecture in Multimedia Processors such as Intel Pentium MMX. None of these halftoning algorithms have used FPGA technology to implement the Serpentine

Scan based Blue-Noise Multitoning with Stacked Error Diffusion algorithm/technique presented in this thesis.

As an example of a parallel architecture implemented into FPGA technology that uses line raster scan to implement an Error Diffusion based halftoning algorithm, an architecture proposed by Christopher Brown and Andreas Savakis [18] will be considered. They discuss the inter-pixel data dependencies, memory requirements and hindrances to parallel processing introduced by their error diffusion technique. They proposed a high performance hardware architecture which exploits multiprocessing to overcome the disadvantages faced during halftoning using error diffusion. The main idea in their approach is to concurrently process pixels in separate rows and columns by eliminating the data dependencies across the processing elements. Their hardware architecture is a high performance color error diffusion image processor realized using FPGA technology. Their Error Diffusion algorithm implemented the basic error diffusion technique invented by Floyd-Steinberg [2] and used a line raster scan technique as shown in Figure 1.3. The input pixel size is 24 bits and the processor gives a 3-bit output. The architecture uses four processing elements allocating one processing core per row thus the first four rows can be processed simultaneously but with some lag between the processing elements. The processor can support a resolution up to 600 dpi resulting in a maximum image size of (5100x6600) which equals 33,660,000 pixels. The design goal is to increase the speed at which the pixels are processed at minimum system cost. An output is obtained every clock cycle and all the processors run on different rows and columns at any point of time. The entire system runs at 80 MHz thus each processing element takes 50ns to complete each computation. Since the algorithm is a straight-forward approach to Floyd-Steinberg's error diffusion, the quality may have been compromised giving rise to artifacts which in turn degrades the whole output image.

In the research and development presented here, an efficient and economical approach towards designing a high performance hardware architecture for serpentine scan based blue-noise multitoning using “stacked” error diffusion is proposed, developed and tested

such that the output quality of the halftoned image is an improvement over those resulting from the use of other halftoning algorithms and approaches.

1.3 Objective of the Thesis

The main objectives of this thesis are described as follows.

- To thoroughly study the halftoning algorithm described in 'C' and convert the whole into an equivalent high speed hardware parallel architecture design and implementation without weakening the quality of the output produced by the original algorithm (The hardware is designed so that the image output from the FPGA and the output obtained by running the algorithm in a serial based CPU are both accurate).
- To achieve a significant performance improvement by greatly increasing the execution speed of the algorithm running on a FPGA when compared to a conventional serial based CPU in which the speed achieved is lowest.
- To develop a high performance hardware architecture which exploits parallelism to the maximum extent possible to improve overall processing performance.
- To design, HDL Simulation test and validate the whole system along with all the components required to develop the system.
- To compare and validate the results obtained from the HDL simulation with the results from the original algorithm running on a CPU.
- To suggest future improvement of the architecture related to enhancing processing performance and output image quality.

1.4 Thesis Outline

This thesis consists of ten chapters which deal with everything about the new halftoning algorithm to HDL Simulation testing and validation of the architecture. The current chapter has provided a detailed explanation of the halftoning algorithm used, previous

research work done on FPGA implementation of such algorithms, present research work dealing with the high quality stacked error diffusion algorithm and the objectives of this thesis.

Chapter 2 deals with the Processor design methodology where type of design, technology used, Data arithmetic or representation, and types of processors used are addressed.

Chapter 3 explains the High level system architecture of the entire hardware system and the working of the system.

Chapter 4 gives a detailed explanation of the Input data memory architecture and how the input pixels are handled.

Chapter 5 provides a view of the architecture of the Processor Core, its design, development and operation.

Chapter 6 shows the detailed architecture of the Error Storage System, its uses, design and operation.

Chapter 7 shows the Output Circuit System Architecture where the output from the Processor core is processed accordingly.

Chapter 8 deals with the Controller Architecture, its implementation, various types of controllers and about the control registers used to minimize device utilization.

Chapter 9 provides a performance comparison with conventional a CPU and quality comparison with other algorithms. It provides a conclusion and future work that can be done to further enhance and improve the results of the current research project.

Chapter 2. Processor Design Methodology

2.1 Introduction

This chapter shows the methods and practices used to design the processor system from the lowest level. The basic levels used to describe this architecture are gate level and the register level. Later in this chapter, the chip technology and the Computer-Aided Design (CAD) tools used to implement the design are discussed. The silicon technology and the CAD software used also decides the design methodology. Design methodology is the first step in the process of developing a hardware architecture. A hardware architecture is the collection of tiny and large components that are interconnected to form a bigger system with a special purpose. The combination of different levels of hardware design hierarchy is called a system.

2.2 Gate Level design

Gate level design, called logic level design, is the lowest level used to describe a functional component. It is concerned with binary values confined to two binary digits 0 and 1. The components designed using gate level designs are logic gates, flip flops which in turn results from the combination of several gates, combinational circuits and sequential circuits resulting from the combination of flip flops.

2.3 Register Level Design

Register level or register transfer level is the next level of abstraction to gate level design. Here, bits are grouped into words and the data is processed as chunks. The main component in this level of abstraction is called a register which is used to store words (collection of bits). The components in register level designs include shift registers, counters, storage registers and accumulators. This level is widely used particularly to save the amount of time it requires to design a component or a system. An efficient

combination of gate level design and register level design is implemented in the hardware architecture of this thesis to obtain the best performance possible.

2.4 Target Technology

There are several latest technologies in which the given Error Diffusion algorithm can be implemented. Some of them include Application Specific Integrated Circuit (ASIC), Programmable Logic Devices (PLAs), Complex Programmable Logic Devices (CPLD), General purpose CPU and Field Programmable Gate Array (FPGA). FPGA technology has an edge over all the other technologies mentioned. FPGA's are flexible, reusable, reprogrammable, cost effective and have the highest possibility of parallel processing. They contain millions of tiny logic blocks with flip flops and have special routing resources to implement a component or a functionality very efficiently. The main difference between a conventional microprocessor and an FPGA is that the microprocessor executes a program in a sequential manner but on the other hand FPGA technology can exploit the parallel processing capability to speed up program execution. FPGA's consist of rectangular array of logic cells. A logic cell basically consists of a look-up-table, a D flip-flop and a 2-to-1 multiplexer. The basic idea behind this technology is that a memory element can implement any type of combinational and sequential function of a size proportional to the memory size. Look-Up-Tables (LUTs) are referred to as memory elements and can be 3 input, 4 input, or 6 input tables. In this research, a Xilinx Virtex-5 (5VFX70TFF665) commonly used FPGA chip [21], Xilinx ISE 10.3 CAD tool [19] and ModelSim 6.4a [20] is used for developing the architecture.

2.4.1 Xilinx Virtex-5 FPGA

Virtex-5 FX FPGA [21] provides the advanced technology for high performance embedded systems along with serial connectivity. It contains many hard/soft Intellectual Property cores, Block RAM's, second generation Xtreme DSP slices, Digital Clock Managers (DCM), Phase-Lock-Loop (PLL) clock generators, Distributed RAM's, 6 input look-up-tables(LUT) and a hard core PowerPC processor embedded inside the chip fabric. The main reason for choosing this version of Virtex device is that the amount of

resources required by the algorithm is satisfied by the 5 series. Table 2.1 shows the features provided by the virtex-5 devices. A hardware description language is used to describe the digital components and program the same on FPGA. In this thesis, Verilog HDL is used to design and develop the required digital logic circuits. Xilinx ISE is a CAD software used to synthesize and implement the verilog code on chip. ModelSim is used in this research to simulate the ISE translated design. The algorithm requires about 3527 Kb of Block RAM space to store the errors and input data generated. Table 2.2 shows the requirements of the current halftoning algorithm.

Table 2.1: Virtex-5 Specifications

Device	Logic Array Size	Slices	Distributed RAM (Kb)	DSP 48E Slices	Block RAM (Kb)	PowerPC Blocks	User I/O
XC5VFX70T	160 x 38	11200	820	128	5328	1	640

Table 2.2: Algorithm Requirements

Device	Slices	DSP 48E Slices	Block RAM (Kb)	User I/O
XC5VFX70T	1616	108	3906	50

2.5 **Data Representation**

Data can be represented in a processor in two ways, namely Fixed point representation and Floating point representation. In this research work, Fixed point arithmetic is used in order to achieve maximum throughput and increase the execution rate by decreasing the execution time required. Floating point arithmetic requires a dedicated hardware unit and consumes a lot of resources. Since FPGA's have a limited number of resources and the clock speed at which the whole system runs is less when compared to a traditional CPU, a Fixed Point Arithmetic is implemented.

2.5.1 **Floating Point Arithmetic**

Floating point [22] is an arithmetic in which the decimal point can be present anywhere in a number. It is also used to represent numbers that would be too wide or too small to fit in computer hardware. The floating point numbers should be normalized to a specific form which helps in simplification of data exchange, floating point algorithms and increases the data storage accuracy. In other words, a floating point number needs to be normalized to a base notation. Normalization says that leading 0's is unacceptable in floating point format. Floating-Point representation has three main fields called Sign, Fraction and Exponent. ***Exponent*** and ***Fraction*** are the two main terms in Floating-Point calculations. ***Exponent*** is defined as the number of times a digit has to be multiplied by itself and ***Fraction (mantissa)*** in hardware terms is a value that that lies between 0 and 1. Figure 2.1 and Figure 2.2 shows the representation of floating point numbers (32 and 64 Bits) in hardware.

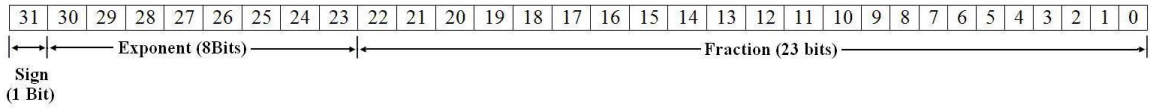

Figure 2.1: 32 Bit Single Precision Floating-Point Representation

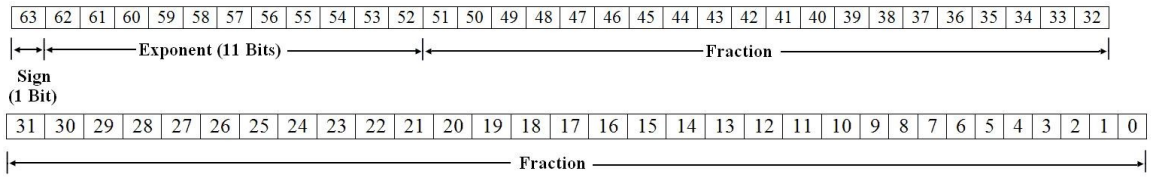

Figure 2.2: 64 Bit Double Precision Floating-Point Representation

The 32 bit architecture provides 23 bits for the fraction, 8 bits for the exponent and 1 bit for the sign. The number of bits allocated for fraction and exponent depends on two main factors namely precision and range. ***Precision*** is the number of binary bits used to represent a particular value in a hardware domain. ***Range*** is the difference between the largest positive number and the largest negative number that can be represented in a given format. Thus, precision of a fraction increases by increasing the number of bits allocated for the fractional part and the range of numbers that can be represented increases by increasing the number of bits in the exponent part. There is a possibility of two

exceptions in a Floating-Point format namely ***Overflow*** and ***Underflow***. ***Overflow*** is an exception in which a positive exponent is too wide to be accommodated in the exponent field. ***Underflow*** is an exception in which a negative exponent is too wide to be accommodated in the exponent field. To overcome these exceptions there exists a choice between double precision (64 bits) and single precision (32 bits). If a system deals with the values that exceed the expectations of a 32 bit hardware, it needs to switch to a double precision format of 64 bits. The original Error Diffusion algorithm addressed in this thesis, written in 'C', uses Floating-Point single precision format and this is one of the reasons for a very high execution time when run on a single CPU. Thus, a Floating-Point format would require at-least 32 bits for representing the data in the algorithm which in turn requires substantial FPGA resources if implemented into FPGA technology and other complex hardware components.

There are four main arithmetic operations performed in any algorithm, they are Addition, Subtraction, Multiplication and Division. Floating-Point operations requires the operands to be normalized before any of the arithmetic takes place. Firstly, the exponents of the two operands should be compared and the smaller operand should be shifted to match the larger operand. Secondly, perform the operation (Addition, Subtraction, Multiplication or Division) on the significands. Thirdly, the result of the operation should again be normalized by shifting and varying the exponent. Finally, check for overflow or underflow and set the appropriate hardware bit if detected. Floating-Point multiplication requires more hardware when compared to other operations as the end result will the twice the length of the operand. The length of the result required depends upon the algorithm, so if an algorithm requires more accuracy, it will use the whole end result but, if the algorithm is not so constrained to accuracy, it uses a part of the results by using techniques called ***Truncation*** and ***Rounding***. ***Truncation*** is the process of truncating or cutting off a required number of digits after the decimal point. ***Rounding*** is the process of approximating a real value to an equivalent simpler value compromising the accuracy to the smallest extent possible. Thus, the choice between rounding and truncation depends solely on the application.

2.5.2 Fixed Point Arithmetic

Fixed-Point [23], [24], [25] is a computer arithmetic in which all the data is represented in integer format. In other words, the decimal point in a real data doesn't vary unlike Floating-Point format. Fixed-Point format supports only a narrow range of values and the hardware required to implement the format is simple. The main concepts used in choosing a Fixed-Point format are Q-Format, Precision, Resolution, Range, Dynamic Range and Accuracy. Both integers and fractions can be represented in fixed point format. Fixed-Point format is used to represent both signed and unsigned data. In this thesis, Fractional Fixed-Point format is chosen to match the original Floating-Point data in the algorithm. Fractional Fixed-Point representation is chosen because it is most suitable for Digital Signal Processing algorithms as the one used in this research work. The range of numbers that the fractional format represents is between -1 and 1. The same is the case in the research algorithm used in this thesis where values never go beyond -1 and 1. ***Q-Format*** is a scheme in Fractional Fixed-Point format used to represent fractions bounded by a fixed binary word length where Q indicates the number of bits used to represent the fraction. ***Precision*** is defined as the number of bits used to represent a data value in a binary or digital world. ***Precision*** is equal to the total word length. ***Resolution*** is the least non-zero value or magnitude which can be represented using a particular Q-Format. ***Range*** is the difference between the maximum positive number and the least negative number represented which ultimately depends on the Q-Format. ***Dynamic Range*** is the ratio of maximum absolute value and the minimum absolute value that can be represented using a specific format. ***Accuracy*** is the magnitude of the difference between a real data value and it's equivalent representation. Due to the extra cost of implementing a dedicated hardware unit for Floating-Point calculations, the Error Diffusion algorithm in this thesis is implemented using a Fixed-Point format that results in significant improvement in throughput, execution speed and reduced hardware complexity. Thus, to improve the execution speed to achieve the best performance, some considerations have to be made before using the Fixed-Point format. Let *Q[I].[F]* be the Q-Format representation of a data value in Fixed-Point where *Q[I]* is the number of bits used to represent the integer part of a value and *Q[F]* is the number of bits used to

represent the fractional part of the value. The sum of *Q[I]* and *Q[F]* yields the total number of bits also called word length and a sign bit at the most significant location used to represent the whole data value. The hardware architecture in this thesis uses *Q[1][14]* format as shown in the Figure 2.3. Thus, the total number of bits equals 16 bits out of which the most significant bit is reserved for the sign bit and the bits after the sign bit location are reserved for representing the equivalent Fixed-Point value.

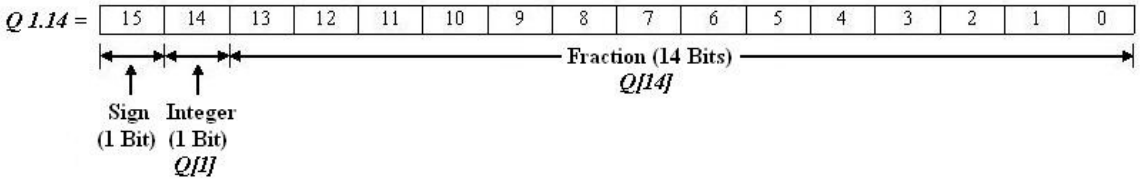

Figure 2.3: Q 1.14, 16 Bit Fixed-Point Representation

Figure 2.3 shows that the word length used is 16 bits or 2 bytes and the dynamic range of the signed integers that can be represented using 16 bits is -32,768 to 32,767. The resolution *r* of the Fixed-Point format is determined by the number of bits used in the fractional part and is shown in Equation 2.1. The maximum positive value (P_{max}) and minimum negative value (N_{min}) that can be represented using this format is shown in equations 2.2 and 2.3.

$$r = \frac{1}{2^F} = \frac{1}{2^{14}} = 0.00006103515625 \qquad (2.1)$$

$$P_{max} = \frac{2^{16-1}-1}{2^{14}} = \frac{32767}{16384} = 1.99993896484375 \qquad (2.2)$$

$$N_{min} = \frac{2^{16-1}}{2^{14}} = \frac{-32768}{16384} = -2.0 \qquad (2.3)$$

Equation 2.4 shows the formula to convert a Floating-Point number (N_{float}) to an equivalent Fixed-Point number (N_{fixd}). The same Equation is used to convert all the Floating-Point data values to Fixed-Point values in this thesis work. The values obtained after the decimal point are truncated as there is no significant deviation observed from the real value. For example if N_{float} is 0.73948, N_{fixd} becomes 12115.6402 and truncating the

result gives the Fixed-Point value 12115 as shown in Equation 2.5. Equation 2.6 shows the Fixed-Point value again converted to Floating-Point to show the accuracy of the conversion.

$$N_{fixd} = N_{float} * 2^{14} \quad (2.4)$$

$$N_{fixd} = 0.73948 * 2^{14} = 12115.6402 = 12115 \quad (2.5)$$

$$N_{float} = \frac{12115}{2^{14}} = 0.73944091796875 \quad (2.6)$$

The difference exists in the fifth decimal point which is about 0.00004 units. Thus, the algorithm in study is not constrained to the accuracy and a meager deviation is allowable. Arithmetic in Fixed-Point is the same as integer arithmetic with minor modifications to multiplication and division operations. As the data being processed lies strictly between -1 and 1, the exception of overflow doesn't occur in this algorithm. The addition and subtraction operations are the same as integer arithmetic. Considering the multiplication operation, if two fractions are multiplied then the resulting fraction will always result in a fraction that will be less than or equal to the two fractions multiplied. But in case of Fixed-Point multiplication, the Floating-Point numbers are first converted to Fixed-Point by multiplying the real value with 2^F where F is the number of bits used to represent the fractional part. As a result, each the fraction number is multiplied by 2^F that gives an incorrect result in Fixed-Point format as shown in Equation 2.7. This can be corrected by dividing the final result by 2^F which will scale it back to Fixed-Point format as shown in the Equation 2.8. Let $\{N1_{fixd}, N2_{fixd}\}$ be the fraction numbers in Fixed-Point format, $\{N1_{float}, N2_{float}\}$ be the fraction numbers in Floating-Point format and $\{N_{fixdm}, N_{floatm}\}$ be the result after multiplication in Fixed-Point and Floating-Point formats respectively. Equation 2.4 is applied to Equation 2.9 for Floating-Point to Fixed-Point conversion and the end result is shown in Equation 2.10. Hence the results in equations 2.8 and 2.10 are equal generating the correct output.

$$N_{fixdm} = N1_{fixd} * N2_{fixd} = 8192 * 8192 = 67108864 \quad (2.7)$$

$$N_{fixdm} = \frac{N1_{fixd} * N2_{fixd}}{2^{14}} = \frac{8192*8192}{16384} = 4096 \qquad (2.8)$$

$$N_{floatm} = N1_{floatd} * N2_{floatd} = 0.5*0.5 = 0.25 \qquad (2.9)$$

$$N_{fixdm} = 0.25*2^{14} = 4096 \qquad (2.10)$$

In the case of Fixed-Point division, the final result should be multiplied by 2^F. Equations 2.11, 2.12, 2.13 and 2.14 show the division algorithm in Fixed-Point format. $\{N_{fixdd}, N_{floatd}\}$ be the result obtained after division in Fixed-Point and Floating-Point formats.

$$N_{fixdd} = \frac{N1_{fixd}}{N2_{fixd}} = \frac{8192}{8192} = 1 \qquad (2.11)$$

$$N_{fixdd} = \frac{N1_{fixd}}{N2_{fixd}}*2^{14} = \frac{8192}{8192}*16384 = 16384 \qquad (2.12)$$

$$N_{floatd} = N1_{floatd} * N2_{floatd} = \frac{0.5}{0.5} = 1 \qquad (2.13)$$

$$N_{fixdd} = 1*2^{14} = 16384 \qquad (2.14)$$

Multiplication produces a result which will be lesser than or equal to twice the width of the operands. The end result can be truncated or rounded depending upon the application requirements. The division by 2^F in Fixed-Point multiplication is achieved by arithmetic left shift and the multiplication by 2^F in Fixed-Point division is achieved by arithmetic right shift. In this research algorithm, Fixed-Point division is not used but all the other arithmetic operations such as addition, subtraction and multiplication are used. Multiplication in this thesis truncates the digits that are not required and uses only the 16 bit output of the multiplied result as shown in Figure 2.4. Bits 29 down to 14 are the only useful bits in this hardware multiplication architecture. There is only a minor loss of accuracy which is not significant.

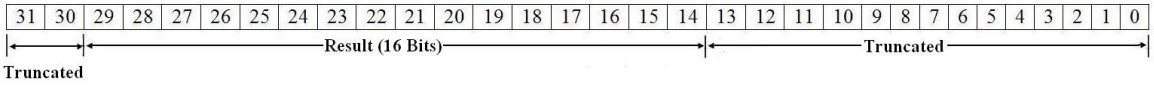

Figure 2.4: Fixed-Point Multiplication End Result

2.6 Types of Processors

Processors are designed based on the application to be executed. There are mainly two types of processing systems, namely General Purpose Processors and Special Purpose Processors. The details about them are discussed briefly in the following sections below.

2.6.1 General Purpose Processors

General purpose processors are normally Programmable Processing systems that are used in conventional computer systems to perform various tasks. Almost all the tasks required by an end user can be run on a general purpose system like Floating-Point operations, Integer arithmetic, external memory interface, general purpose I/O, signal processing and control of other devices, etc. These processors are fast, but sometimes not very suitable to run a specific application where parallelism and execution time are important. A Programmable Processors Instruction Set Architecture (ISA) is a command set architecture which tells the processor what to do with the data available at any instant of time. For example, RISC is a Reduced Instruction Set Computer of length 32 bits and CISC is a Complex Instruction Set Computer ISA which has more complex instructions when compared to RISC.

2.6.2 Special Purpose Processors

Special Purpose Processors, as the name suggests, are developed and used for specific applications. This research work deals with a specific halftoning algorithm where the same set of operations are repeated periodically. Thus, there is no need of an Instruction Set Architecture where the processor needs to know what operation to perform. The operations are hardwired in this type of application and there are dedicated control units to execute the algorithm. The reasons behind choosing a Special Purpose Processor is that the cost of making the chip is sometimes less for a specific purpose, the number of silicon gates required depends upon the size of the algorithm and the execution speed can be drastically increased due to exploitation of parallel processing in the algorithm. In this thesis, an algorithm specific hardware parallel processor architecture is designed and described using the Verilog Hardware Description Language. The architecture is simulated, tested and validated using the ModelSim Simulation CAD tool.

Chapter 3. High Level System Architecture

3.1 Introduction

This chapter deals with describing the entire system architecture which implements the halftoning algorithm and describes a process flow diagram showing the functional operation of the architecture as it implements the halftoning algorithm. As discussed earlier in Chapter 1., this thesis deals with a special purpose high performance processing system which can efficiently execute a proprietary halftoning algorithm at the maximum processing speed possible. Unlike general purpose processing systems, this architecture does not need an Instruction Set Architecture but has a predefined set of instructions that will run for every pixel that is processed. The concept of pipelining cannot be applied here as inter pixel dependencies tend to be the highest. At any given point of time, a pixel cannot be processed unless the preceding pixel is fully processed. Parallelism is introduced in this system where different pixels can be processed in parallel. Thus, the 'C' version of the algorithm is broken down into segments and the areas where the code can be parallelized are determined. Equivalent hardware units are designed, tested and fully verified before combining the units to form a larger system. Each of the combined system functional units obtained by connecting the individual functional units of a system is also rigorously tested for discrepancies and fixed if errors are found. The high level system can also be referred to as a CPU (Central Processing Unit) which consists of processing elements, memory elements and output logic. The system implements a sequence of microoperations resulting in an output desired by the application. A system consists of the data handling unit or the processing unit, control unit or controller and interfacing units to communicate with other devices outside the chip. The processing part of the system is also called the *datapath* of the CPU where input data is processed accordingly. The *datapath* of a CPU consists of many smaller digital units namely multiplexers, registers, decoders which in turn are built from a lower level components called *gates*. *Gates* are derived from the basic component *transistor*. The *control unit* is one of the major components in a digital system which is responsible for the correct behavior of a

circuit. So, a control unit controls the *datapath* and other components which constitutes the processing system. The *High Level System Architecture* of a system is defined as the abstraction of lower level components by just describing the function or behavior of the system. The following sections give a very detailed explanation on the behavior of the hardware functional architecture.

3.2 High Level System Hardware Architecture

This section gives a detailed description of the hardware architecture of the unit that implements the Stacked Error Diffusion Halftoning algorithm. It shows the organization and operation of the whole system. Figure 3.1 shows the high level system architecture design that includes five main modules, namely a Host PC, DDR2 RAM, Flash ROM, FPGA and a Printer. The first module is the host PC which serves as the source from which the input image pixels are buffered. Here, the image to be halftoned is read in an interleaved format. Interleaved format of an image is the bundle of all the channels in one pixel followed by the next pixel with all the channels packed. For example, if Cyan, Magenta, Yellow and Key are the four channels in a pixel then, CMYK of the first pixel will be packed together, CMYK of the second pixel and so on. The PC reads the input image as $[CMYK]_1$,$[CMYK]_2$,$[CMYK]_3$ ..... $[CMYK]_n$ where $'n'$ is the number of pixels in the image. The data width of input pixels supported by the architecture is 8 or 12 bits per channel per color. The input data from the pixel is extracted and is buffered to the **DDR2 RAM** (Double Data Rate Random Access Memory) for subsequent processing with the help of any high speed interface preferably PCIe (Peripheral Component Interconnect Express). PCIe is chosen to match the speed of the FPGA and the DDR2 memory as there should not be any delay in buffering which affects the performance adversely. The second module comprises of a **DDR2 RAM** which will be filled with at least two consecutive rows to prevent buffering problems. The third module is the **Flash ROM** (Read Only Memory) used to store the bit stream file of the architecture generated which is used to program the FPGA. As the FPGA is a volatile memory semiconductor device, it needs an external flash memory to store the hardware architecture and initialize

itself at system power-up. The fourth and imperative module is the FPGA which the hardware architecture described in **Verilog HDL** is programmed into.

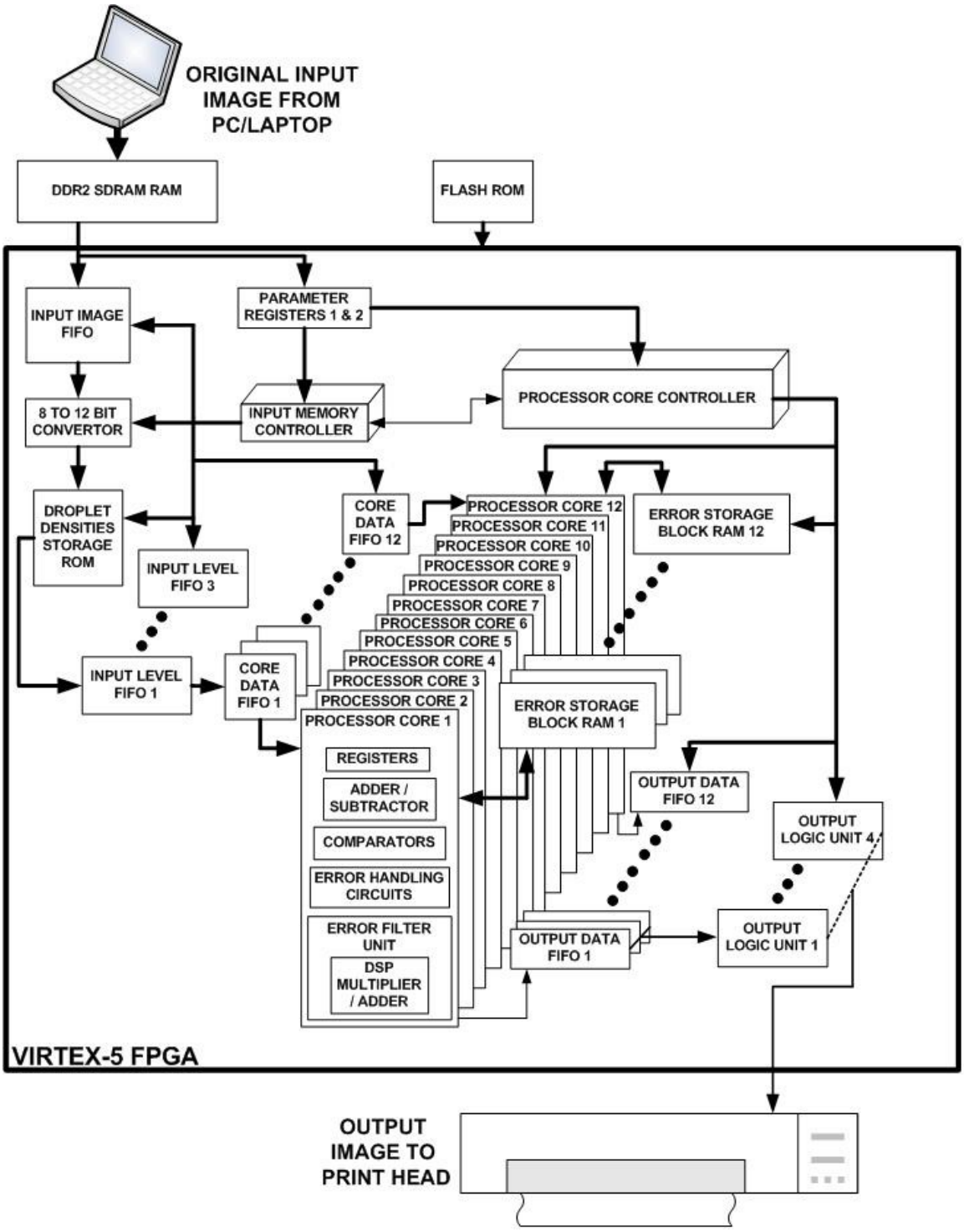

Figure 3.1: High Level System Hardware Architecture

The halftoning algorithm is examined and areas where the code can be parallelized are determined. The algorithm uses the Stacked Error Diffusion technique which increases the inter-pixel dependencies and as the Stacked Diffusion implies, each level in a pixel depends upon the previous level of the same pixel. Thus, the pixel cannot be fully processed without processing all the levels and there is no dependency among the colors (inks or channels). The colors in the pixels can be processed in parallel and the only constraint that exists is between the levels in each color of a given pixel. Another major hurdle to parallel implementation is that the pixels are scanned in a serpentine fashion, this amplifies the memory demand as the errors being diffused at each pixel needs to be stored until the pixel to which the errors are dispersed is processed. Thus, the amount of memory required is directly proportional to the width of the input image. Taking all the dependencies into account, the following factors decide the implementation of the algorithm in hand on a FPGA.

- Each level in every channel is treated as an individual processing unit. So the number of channels and the levels in each color decides the number of processing cores to be used for executing the algorithm.
- The errors dispersed in each pixel location should be stored in a data memory which can be accessed later. Resolution is defined as the number of dots that can be printed per inch of an image. The current algorithm supports a resolution of 720 dpi (dots per inch), maximum width of 24 inches and the error data is 16 bits wide. Thus the memory required to store these errors will be 24 * 720 * 16 which is approximately 276480 bits (33.75 KB) per processing core.
- As discussed earlier in section 1.1.7 , droplet densities are to be stored in look-up tables. This requires data memory that depends on the number of levels used in the original image. For example, if 3 levels are used and the original image data input is 12 bits, the amount of storage locations required is $3 * 2^{12} = 12288$. The total amount of memory required if each data value to be stored is 16 bits will be 12288 * 16 = 196608 bits (24 KB).

- The current hardware architecture is designed to handle up to 4 channels and 3 levels per channel. The different combinations possible in this design are CORE[*c,l*] = {[4,3],[4,2],[4,1],[3,3],[3,2],[3,1],[2,3],[2,2],[2,1],[1,3],[1,2],[1,1]} where *c, l* is the number of channels and number of levels per channel respectively. The design is flexible and can be extended depending on the printer configuration and the input image depth (Number of bits per pixel).

- The architecture implements fixed point arithmetic with Q1.14 format where the calculations are done the same way as in integer arithmetic. The main reason for using Fixed-Point arithmetic instead of Floating-Point is that the floating point calculations requires complex calculations, takes a lot of resources and consumes more time to process a pixel when compared to Fixed-Point arithmetic. Also, there is no degradation observed in the output image when this format is used when compared to a Floating-Point format.

- The number of hardware units or logic resources used by the algorithm is described as follows. The architecture requires 48 high speed multipliers (Xtreme DSP (5.2)), 60 high speed adders (Xtreme DSP (5.2)), 405 KB of data memory for storing the errors generated, 24 KB of storage memory for the loop-up tables and 2 KB of memory for input buffering.

The main functional units in the architecture are the datapath, control unit and the output logic unit. The datapath unit consists of Input Image FIFO, 8 to 12 bit Convertor, Droplet Densities Storage ROM, Input Level FIFOs, Core Data FIFOs, Error Storage Block RAMs, Processor Cores (1-12), Output Data FIFOs, Output Logic Units and the control unit has Input Memory Controller and Processor Core Controller.

3.2.1 **Datapath Architecture**

The datapath of this system has a 16 bit architecture and every register and other storage devices inside the FPGA are 16 bits wide except the Input image FIFO that is 12 bits wide. Figure 3.1 shows the digital functional components connected to each other inside the FPGA. The function of each component is described as follows.

- **Parameter Registers (1 and 2):** There are two Parameter Registers of width 32 bits. Parameter Register 1 is used to store the input Image Size and Parameter Register 2 is used to store the number of Channels, number of Levels and the information about the number of bits required to represent each Channel (8 bits / 12 bits).

- **Input Image FIFO:** The input pixels from the DDR2 RAM are stored inside the FPGA with the help of this FIFO and accessed according to the need of the processing cores. The data from the DDR2 RAM is transferred to the FPGA FIFO with the help of a memory interface running at a speed proportional to the FPGA. The Input Image FIFO is similar to a Distributed/Block RAM inside the FPGA and the only difference is that it increments the address from the top of the stack to its bottom with respect to the read or write command given to the FIFO. Data can be simultaneously written to or read from the memory locations. At the beginning of process start-up, the FPGA will not start processing until the Input Image FIFO is completely filled.

- **8 to 12 Bit Convertor:** The 8 to 12 Bit Convertor is a combinational circuit that is used to convert an 8 bit input value to 12 bits with the help of Padding Technique. Thus, the halftoning architecture can support 8 or 12 bits per channel as shown in Figure 3.1.

- **Droplet Densities Storage ROM:** Once the Input Image FIFO of Figure 3.1 is filled, the pixels are buffered through a Droplet Densities Storage ROM also called Look-Up-Tables (LUT) which is a Read only memory that has the mappings from gray-level to droplet densities for each pixel value of 12 bits.

- **Input Level FIFOs (1-3):** There are three Input Level FIFOs which are used to buffer the data from the Droplet Densities Storage ROM. The number of Input Level FIFOs used is directly proportional to the number of Levels in each channel or the number of Droplet Densities per channel.

- **Core Data FIFOs (1-12) :** There are 12 data buffering Core Data FIFO memory files, one for each Processor Core of Figure 3.1. It stores the input pixel data to be processed obtained from the LUT array and has 4 address locations each of width 16 bits.

- **Processor Cores (1-12):** There are 12 processing cores in the architecture of Figure 3.1. All the cores are identical and each of them consists of Registers, Adder/Subtractors, Comparators, Error handling circuits and Error Filter circuits containing multipliers and adders. Each core is 1 clock cycle behind the immediate core succeeding it. For example, Processor core 2 is 1 cycle behind Processor core 1, core 3 is 1 clock cycle behind core 2 and core 12 is 12 cycles behind core 1. The registers are used to store input pixel data, previous pixel values, current processed pixel values and neighboring partial pixel values. The Adder / Subtractor unit is used to add the input pixel value with the previous pixel value and to subtract the current output value from the combined value of input pixel. The comparators are used to compare the results and the output of the previous level with a threshold constant making sure that the values satisfy the stacking constraint. The error handling circuit consists of a couple of comparators to make sure that the error values produced are in the range between -1 and 1. The error diffusion unit in each core contains 4 multipliers and 4 adder circuits to calculate the errors. This unit also contains a random weights generator which is used to perturb the weights at each pixel location.

- **Error Storage Block RAMs (1-12) :** There are 12 Error Storage Block RAM memory files, one for each Processor Core of Figure 3.1.The Error Storage Block RAMs store the errors corresponding to the pixel locations. The Block RAM is 16 bits wide and has 17280 memory locations. The number of address locations is calculated based on the maximum width of the image which is 24 inches multiplied by the resolution of 720 dpi.

- **Output Data FIFOs (1-12):** There are 12 Output Data FIFOs, one for each Processor Core shown in Figure 3.1. The outputs from the Processor Cores are stored in this Output Data FIFOs which are 1 bit wide having 2 memory locations per Processor core.
- **Output Logic Units (1-4):** Four Output Logic Units are designed based on the number of colors and levels supported by the architecture. The number of Output Logic Units is directly proportional to the number of channels in the image. The outputs from the output FIFO array are mapped to 2 bits per channel per pixel. The output data of the processed image (Halftoned Image) is sent directly to the print head of a printer with the help of an output interface preferably Ethernet.

3.2.2 Control Unit Architecture

This section deals with the control strategy of the hardware architecture. The system consists of two controllers, one for managing the memory (RAM) elements and the other for controlling the Processor Core operations. Figure 3.1 shows the Input Memory controller used to manage the data buffering operations and Processor Core Controller used to control the operations of the Processor Core, Error Storage Block RAMs, Output Data FIFOs and Output Logic Circuits. The control unit performs the timely execution of predefined micro-instructions to obtain the desired performance and results. The whole datapath is controlled using the two control units mentioned. Detailed explanation of controller functionality and architecture is provided in Chapter 8.

3.3 High Level Process Flow Description

This section provides a detailed explanation of how the hardware system functionally operates. Figure 3.2 shows the flow chart that provides the step by step operational procedure of the halftoning algorithm running on the hardware architecture of Figure 3.1 programmed into the FPGA. The following points discuss the operational flow chart in Figure 3.2.

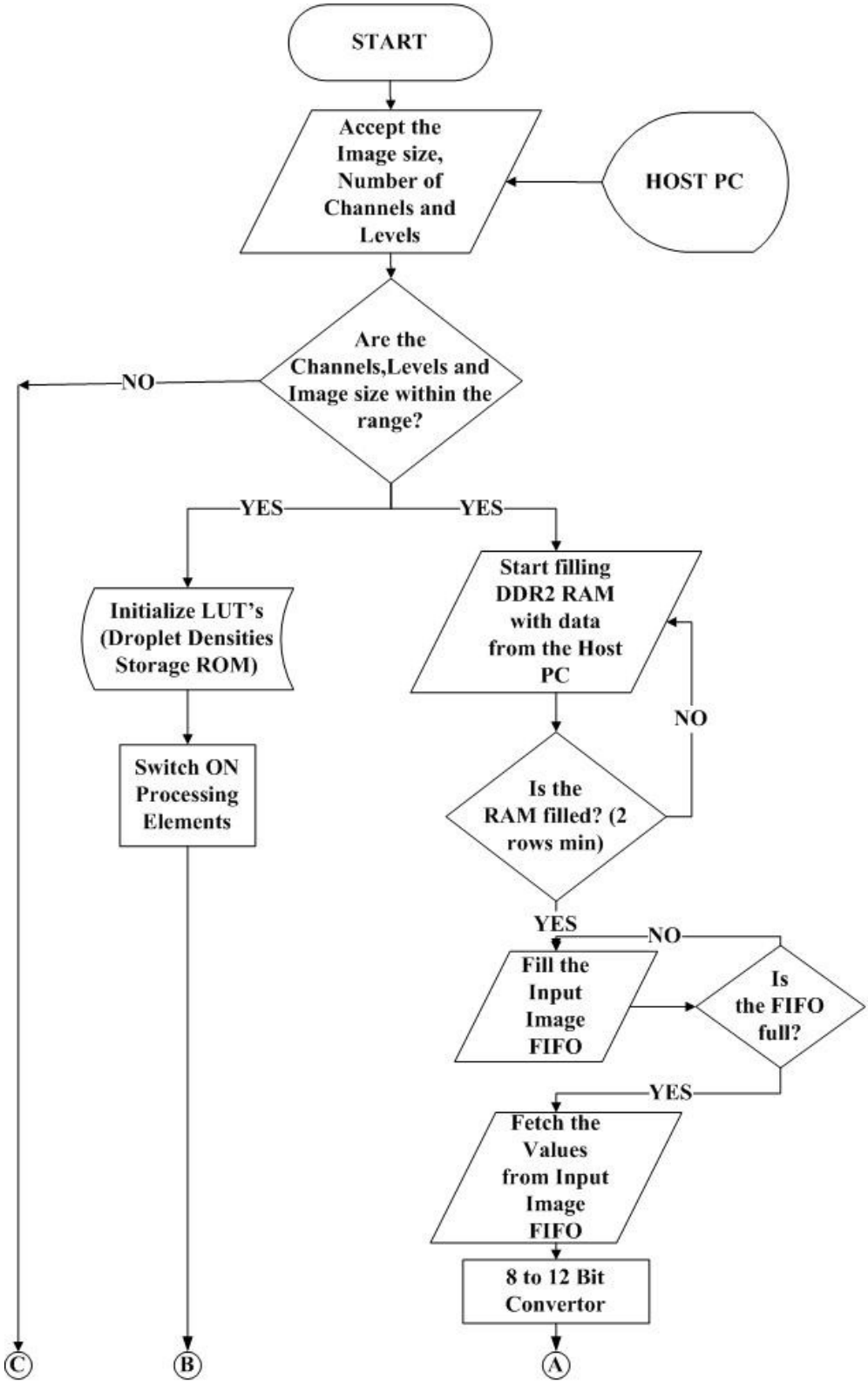

Figure 3.2: Hardware Operational Procedure Flow Chart 1

- FPGA accepts the initial parameters of the image namely image size, number of channels and levels from the host PC.
- If the parameters are within the range of values that is supported by the FPGA, then a 'go' signal is asserted to the DDR2 memory to accept input pixels from the host PC else if the parameters don't match, the system shows an error message stating that the parameters entered are unsupported. The FPGA also initializes the Droplet Storage ROM and the Processor Cores are switched ON according to the number of channels / levels.
- The memory interface checks whether at least 2 rows of data is inside the DDR2 memory. If the RAM is filled, the FPGA starts accepting the input data and fills the Input Image FIFO. The controller inside the FPGA constantly monitors the FIFO and stops filling it when it is full.
- The input pixels are buffered through a 12 bit convertor circuit which converts a 8 bit input value to 12 bits using a *padding* technique.

The following points address the flow chart in Figure 3.3.

- The output from the 12 bit convertor of Figure 3.2 is fed to the Look-up-tables or the Droplet Densities Storage ROM to get the Droplet Size Values (see Figure 3.1) that are associated with a specific input pixel.
- The values obtained from the Droplet Densities Storage ROM are stored in the Input Level FIFOs which in turn is diverted to the Core Data FIFOs specific to each Processor Core as shown in Figure 3.3.
- All the processing elements have the same architecture and hence only one architecture is shown in the flow chart.

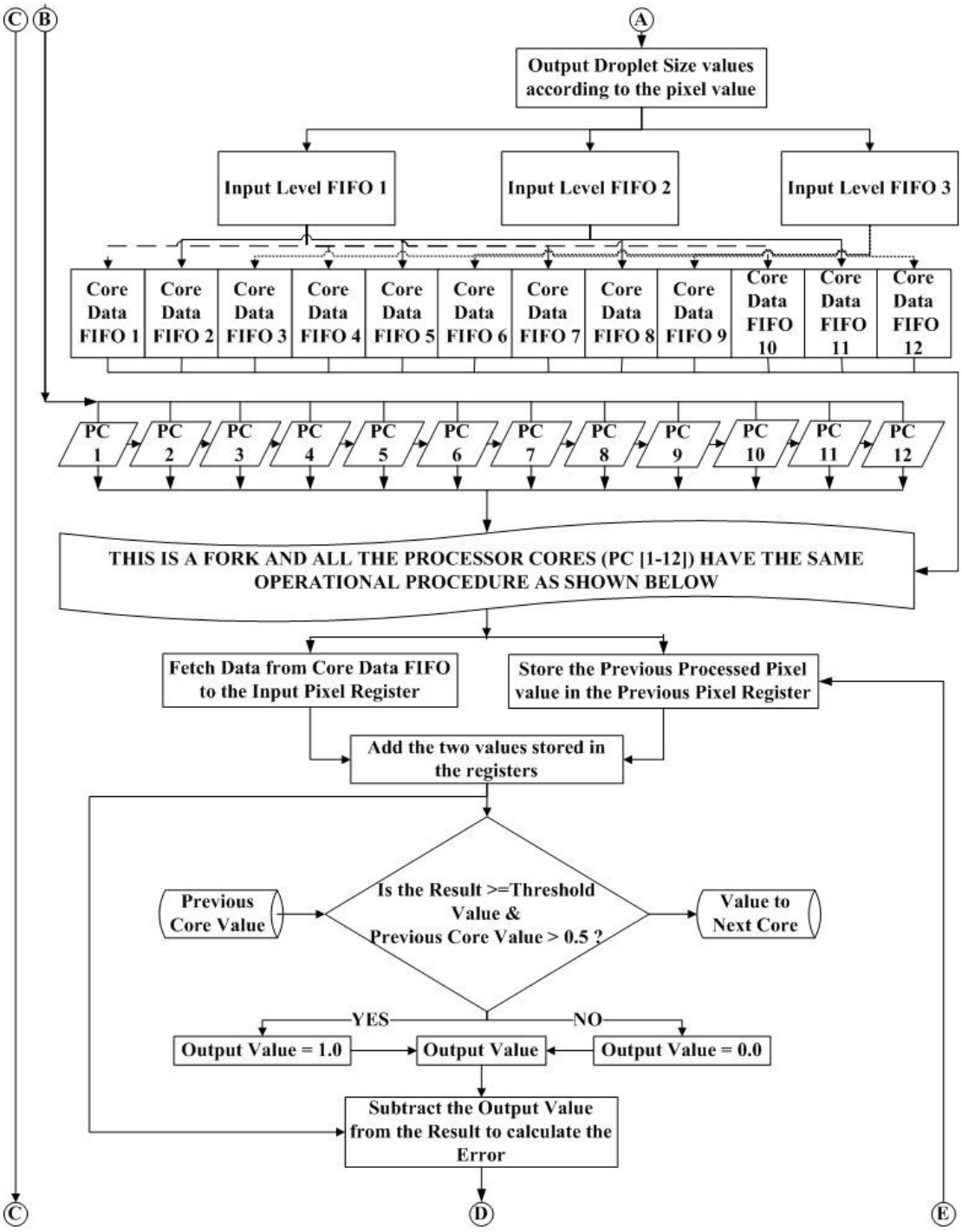

Figure 3.3: Hardware Operational Procedure Flow Chart 2

- Data from the input Data RAM is fetched and stored in the Input Pixel Register. The previous processed pixel value is stored in the Previous Pixel Register. The

two values are added with the help of an Adder-Subtractor unit.

- The result from the unit is compared with a threshold value (0.5 in this algorithm) and the previous output of the level in the same channel is compared with a constant 0.5.
- If the value satisfies the condition, the output value is tied to 1 and if the condition is not satisfied, the output value is tied to 0.
- When the output value is assigned 1 or 0 there exists an error between the result from the adder and the output value. The Error is determined by subtracting the output value from the result of the Adder-Subtractor unit.

The following points address the operational flow chart in Figure 3.5.

- The error value is constantly scanned by an error limiting circuit that assigns 1 if the error is greater than 1 and assigns -1 if the error is less than -1. But, if both conditions aren't satisfied, then the error is within the limits and is stored in the error register.
- The error diffusion circuit comes to play in this step. The controller checks the corresponding pixel that is being processed and manages the operations of the multiplier and the adder circuit according to the Figure 3.4.

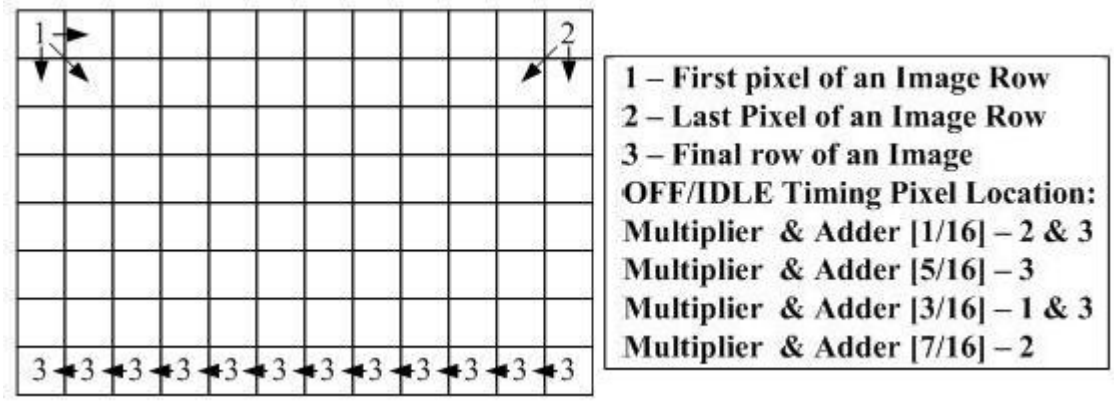

Figure 3.4: OFF / IDLE Timing Pixel Locations

- Figure 3.4 shows the pixel locations where the error filter elements (Multiplier & Adder) are in the OFF state. Figure 1.2 and 1.4 shows the weight distribution and the weights at different pixel locations in a serpentine scan methodology.

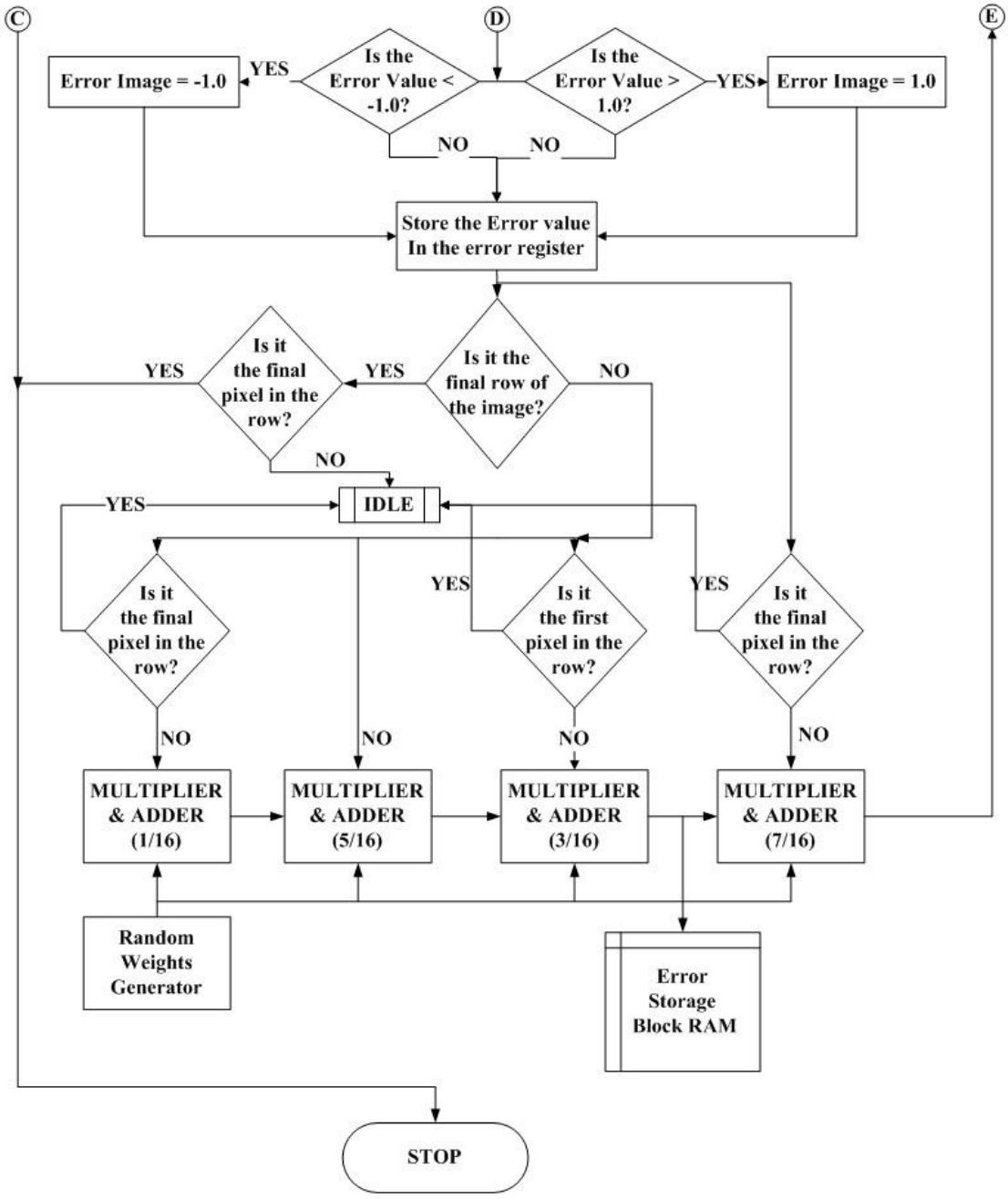

Figure 3.5: Hardware Operational Procedure Flow Chart 3

- The multiplier is used to multiply the error stored in the error register with the random weights produced by the Random Weight Generator. The adder then adds the resultant value from the multiplier with the previously stored error value.
- The number of error values generated per pixel per channel per level is 4 as at almost all the pixel locations, errors are distributed to the four nearest neighbors as shown in Figure 1.2. The error values are stored in the Error Storage Block RAM inside the FPGA for subsequent processing.
- The FPGA stops processing once it reaches the final pixel of the image and goes to an IDLE state where it can be again restarted to process the next upcoming image.

3.4 **Hardware Algorithm Execution**

This section provides information about the Error Diffusion Halftoning algorithm execution in FPGA. Figure 3.6 shows the operational design of the Stacked Error Diffusion algorithm currently being used in this research. The input colors are mapped to the processing cores and halftoned with the help of this algorithm. All the processing cores are connected to their Core Data FIFO input and the output bit of the previous core. As there exists no stacking constraint between the channels, the starting level of each channel is tied to a constant bit of 1. One of the inputs of core 1 is tied to a constant 1 as it is the first level. The succeeding cores are connected to one another i.e. the output of the first core is connected to one of the inputs of the second core; the output of the second core is connected to one of the inputs of the third and so on till it reaches the final core. The output from each individual core is connected to an output logic which calculates the output pixel value. As there is a dependency constraint, all the cores cannot start processing a pixel at the same time. Thus, there exists a unit delay circuit in the design which delays the processing time by one clock cycle compared to the succeeding core. In other words, core '*n*' will not produce the output until it receives the output from core *(n-1)* where *n=1, 2….n*. The pixels are processed in parallel but the first core will be ahead of its previous core by one clock.

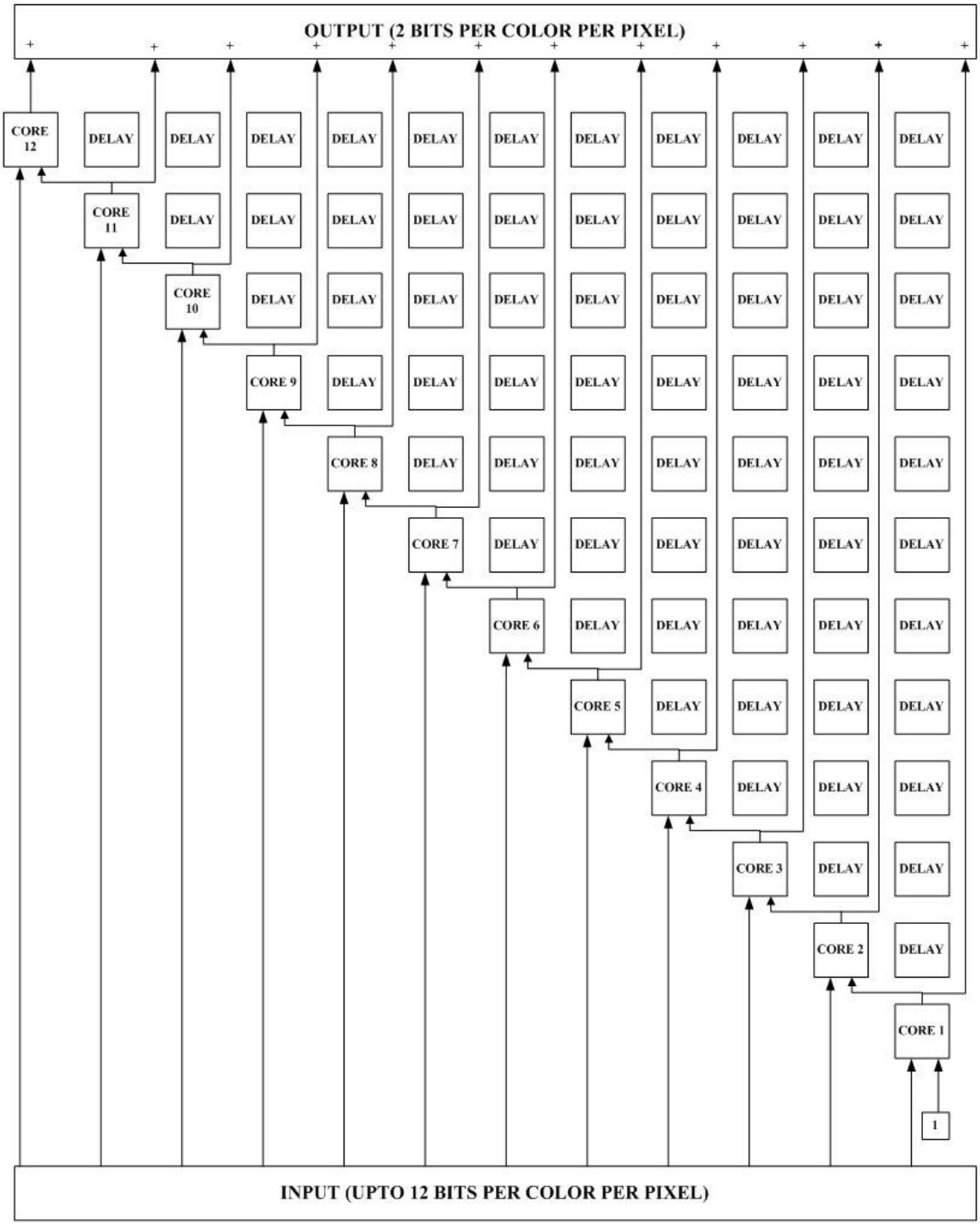

Figure 3.6: Processor Cores Pixel Execution Sequence

Some of these cores can run in parallel at the same time and this parallelism is between the colors or channels in the input image. For example, CMYK can be four channels in an image assuming three levels in each color. Colors C, M, Y, K can be run in parallel since

there is no constraint between different colors. But in this architecture each core is set to run behind its succeeding core as there is no performance difference and it can also support more than 4 colors / 3 levels per color as in the case of this architecture. The whole system is run at 50 MHz to achieve the desired performance and the output is obtained every 8 clock cycles as shown in Figures 3.7 and 3.8 respectively.

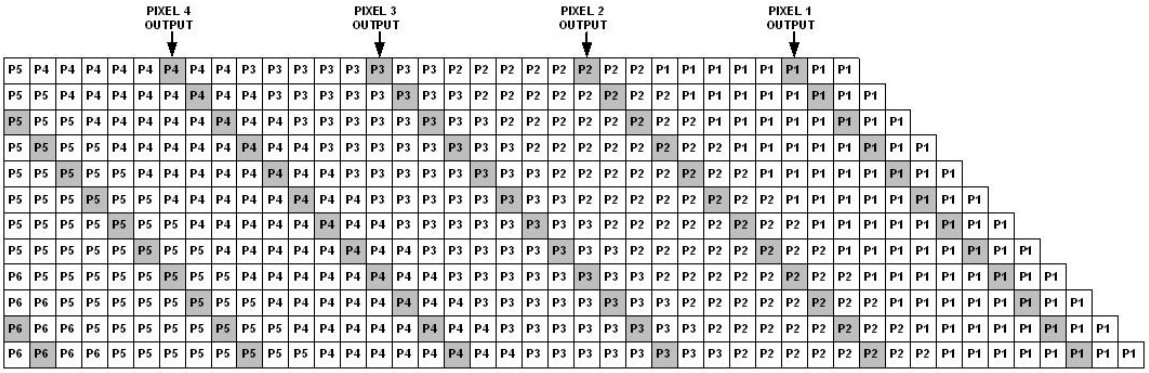

Figure 3.7: Current Hardware Execution Methodology

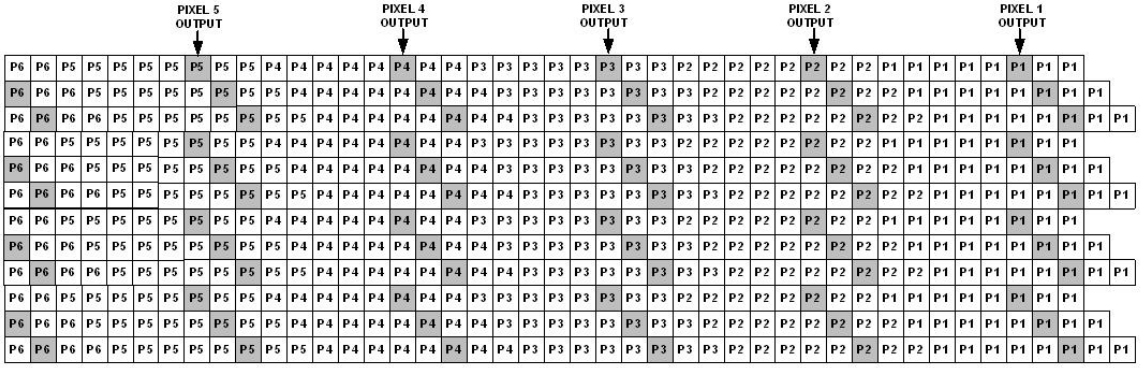

Figure 3.8: Alternate Hardware Execution Methodology

Figure 3.7 shows the pixels being processed under current research methodology where every core is one step behind its succeeding core. There is also an alternate architecture shown in Figure 3.8 where all the colors per pixel are run in parallel and the output is still obtained every 8 clock cycles. Thus, from this research the processing cores gives output every 8 clock cycles irrespective of the number of channels and levels per channels in an image. The current architecture is capable of speeds up to 130 MHz and can be altered according to the printer requirements to achieve a specified throughput. The current architecture is about twice as fast as the modern day wide format printers. Thus this design is scalable, flexible and compatible with any printer configuration.

Chapter 4. Input Data Memory Architecture Design

4.1 Introduction

This chapter provides a comprehensive explanation of the memory systems and the operations performed on different memory RAMs. The input memory architecture is the first and foremost system that the processor core depends on for efficient buffering of image pixels. Calculations are not performed in this segment as it deals primarily with memory storage and access. The chapter discusses 5 major digital elements namely the Input Image FIFO, Parameter Registers, Droplet Densities Storage ROM (LUTs), Input Level FIFO and Input Core Data FIFO. The next section addresses about the Xilinx Virtex-5 FPGA components used to build all the digital memory elements listed above.

4.2 Xilinx Virtex-5 Memory Components

The Virtex-5 devices [26] consists of two main memory components called Block RAM and Distributed RAM. The choice of the components depend upon the memory size and speed requirements. For example, if a design requires buffering of data at high speed and has a small storage constraint then Distributed RAM can be used to meet the expectations. But if a design requires a large storage space inside the chip and an average speed of access then a Block RAM can be used. Both RAM's are Static Random Access Memory (SRAM) systems where the binary bits are stored with the help of internal latches. The main reason for using SRAM is that it has shorter read and write cycles and is faster when compared to other RAM's. An SRAM is a volatile element and loses its contents when the device is powered OFF. Unlike Dynamic RAM's, SRAM's do not require refresh and precharge cycles. Hence SRAM's are faster when compared to DRAM's but not area efficient (consumes more space when compared). The current design under consideration has a requirement of a very large storage space and an optimal speed at which the algorithm should be run. So, the Input Memory Architecture design

incorporates both Distributed and Block RAM's to allow efficient execution of the Stacked Error Diffusion halftoning algorithm.

4.2.1 **Block RAM**

The block RAM in Virtex-5 FPGA (XC5VFX70T) chip can store a maximum of 5328 Kilobits of data. The RAM's can be cascaded and configured for a deeper and wider memory space depending upon the storage requirements. The device supports both synchronous and asynchronous memory operations but, the current design recommends a synchronous operation to avoid timing conflicts in the design. Block RAM can be used as a single port or a dual port memory element depending on the memory access requirements. In this research, both single and dual port elements are used based on the halftoning algorithm. The RAM can also be used as a ROM (Read-Only-Memory) which has a major use in this algorithm implementation. The memory locations in the RAM can be initialized to a predefined value and can be changed during the device operation. The various configurations available in a Block RAM are *Single-Port RAM/ROM, Simple Dual-Port RAM and True-Dual Port RAM/ROM* as shown in Figure 4.1.

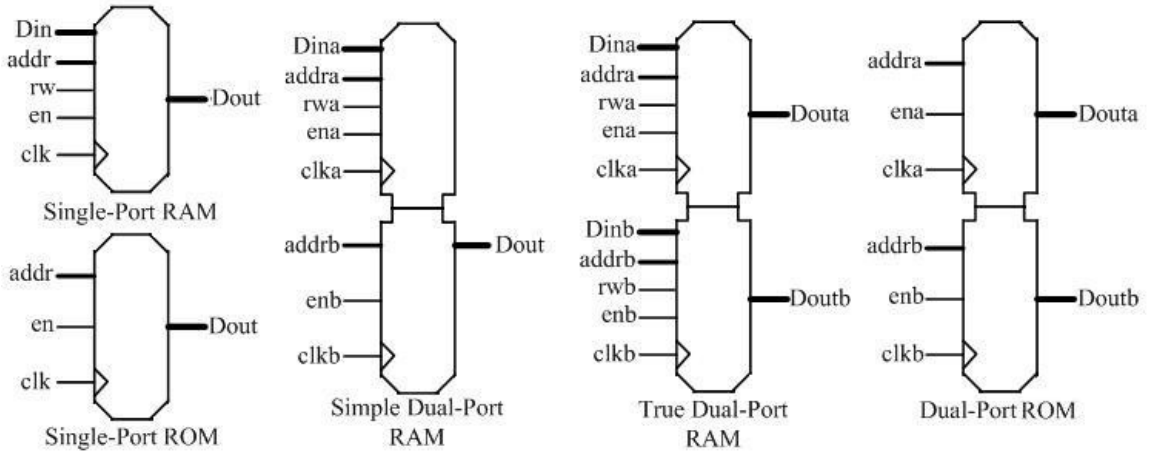

Figure 4.1: Types of Xilinx Virtex-5 RAM / ROM

Single-Port RAM/ROM is the memory storage component where there exists only one data input and data output port. It can be used as a look-up-table for accessing stored values by the processor. A *Simple Dual-Port RAM* is the storage element where there exists one data input port and only one data output port. This can be typically used in

scenarios where a processor needs to read from one location and write to another location in the same RAM simultaneously. A *True Dual-Port RAM/ROM* is a type of memory storage where more than one processing element with different read and write locations wants to access the same storage element. It contains two data input ports and two data output ports. The parameters in the Figure 4.1, namely *addra, addrb,* are the address values that can be provided to access a particular location in the memory. *rw, rwa* and *rwb* are the read-write controls signal used to select either read or write operation (typically '0' indicates read and '1' indicates write); *en, ena, enb* are the enable signals that is used to control the chip select operation (basically '0' means a chip can be used for memory operations and '1' means that the chip cannot be used), *clk, clka, clkb* are the clock ports for synchronous operation of the memory devices and *din, dina, dinb, dout, douta, doutb* are the inputs and outputs of the corresponding memory blocks. There exists three operating modes for the Block RAMs that regulates the read and write behavior of the ports. The operating modes are W*rite First mode, Read First mode* and *No Change mode*. In *Write-First mode*, the input data is written to the memory location and the data written is reflected at the output simultaneously. In the *Read-First mode*, the input data is written to the memory location whereas the previous stored data is reflected at the output. This mode is also called *Read-before-Write*. In *No-Change mode*, the data at the output port reflects the same data from the previous read operation and is unaffected by the current write operation. In this research, the *No Change mode* is used and the hardware is designed in such a way that there exists no conflicts and collisions. The read and write operations require one clock edge to provide the output. Large FIFO's (First-In First-Out) can be instantiated using Block RAMs. Performance upto 450 MHz can be obtained using the Block RAM module embedded inside the FPGA. But the current research work doesn't need such high speeds and hence the memory modules are run at a speed required by the application.

4.2.2 **Distributed RAM**

In addition to the Block RAMs, there are Distributed RAMs embedded throughout the FPGA chip. This RAM is very fast, available at all the regions inside the FPGA and is

optimal for high speed data buffering, small FIFO's and can be used as register files. The Distributed RAM has all the features as the Block RAM such as *Single-Port RAM/ROM*, *Dual-Port RAM/ROM*. The only disadvantage of these RAMs is that the memory availability is much less compared to the Block RAM module. The Virtex-5 provides a maximum of approximately 820 Kilobits of storage space in terms of distributed RAM. Synchronous and Asynchronous operations can be performed efficiently on these RAMs where write operation is typically synchronous and read is asynchronous. One can also program the RAMs to perform fully synchronous behavior depending on the application need. The RAM memory can be initialized with some values and can also be changed during the device operation. Thus a full flexibility in design is allowed which is similar in the case of Block RAMs.

4.3 **Xilinx Core Generator**

The CORE Generator [27] in Xilinx is a proprietary design tool that instantiates Intellectual Property (IP) modules which can run very efficiently on the Xilinx FPGAs. The Core Generator provides functional digital elements such as FIFOs and memories (both Block RAMs and Distributed RAMs), Multipliers (Xtreme DSP and LUT based), Adders-Subtractors, Standard Bus Interfaces, Memory interfaces, Comparators, Counters, Shift-Registers and Dividers. In this research, Core Generator modules are used at places where the performance of the device is crucial. This results in less time consumed in designing the hardware since it takes an ample amount of time to design a module like an adder from the scratch and to test it and the modules provided by the Core Generator are very efficient and fully tested. All the designs in this research project are described using verilog and tested using the ModelSim simulation tool.

4.4 **Input Image FIFO**

The first memory element in this hardware design is the Input Image FIFO (see Figure 3.1) which is used to store the input pixels sent by the host PC. This FIFO is generated with the help of the Xilinx Core Generator design tool that utilizes Block RAMs in the

FPGA. The memory element is tailored to meet the requirements of the Halftoning algorithm. The following sections briefly describe the design and operation of the Input Image FIFO.

4.4.1 **Input Image FIFO Design**

The Input Image FIFO is designed with the help of the Core Generator wizard. Verilog HDL is used to describe the design and is fully tested using the ModelSim simulation tool. In order to store the pixels for continuous buffering to the processor cores, the width and depth of the FIFO must be large enough to prevent absence of data at a given time. The width of the FIFO depends on the number of bits used by a channel in a pixel, as this algorithm supports both 8 / 12 bit data, the data width of 12 bits is selected. To prevent buffering discrepancies, the depth of the FIFO is 1024 bits. The depth of the memory element indicates the number of address locations that can accommodate 12 bits per location. So the number of address lines required to access 1024 locations is 10 bits (2^{10} = 1024) starting from address 0 till 1023. From the width and depth of the FIFO, the amount of storage space consumed can be calculated by multiplying the two parameters 12 * 1024 that comes to 12288 bits in total. The difference between a Random Access Memory and a FIFO is that any address location in the RAM can be accessed at any point of time but, in a FIFO there is an internal counter that increments the memory address by a count of 1 depending on the operation (Read/Write) and it acts like a stack where the first element filled should come out first and so on. Figure 4.2 shows the schematic of the image FIFO used in this design.

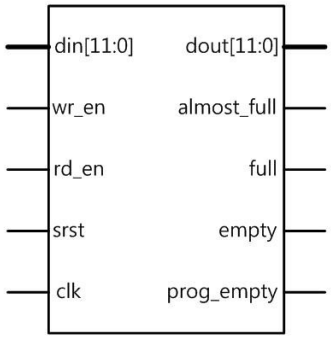

Figure 4.2: Input Image FIFO Schematic

4.4.2 **FIFO Operational Procedure**

The FIFO is designed based on the original halftoning algorithm and a code snippet is shown in Figure 4.3.

```
for (m=0; m<imageRow; m++)
{
  TIFFReadScanline(inTiff, inputBuffer, m, 0); ⟶ 1
  for (n=0; n<imageCol*imageChn; n++)
  {
   inputRowBuffer[n]=((int)inputBuffer[n]<<4)+((int)inputBuffer[n]>>4); ⟶ 2
  }
```

Figure 4.3: Software Code Snippet for Image FIFO and 12 Bit Conversion

The line marked '*1*' in the code snippet indicates that an image is being read into the '*inputBuffer*' unit which in this hardware architecture is the Input Image FIFO. Figure 4.2 shows the input and output pins of the FIFO where *din[11:0]*is the input pixel data of width 8 or 12 bits, *dout[11:0]* is the output of the FIFO which has the same width as the input port, *wr_en* is the write enable port ('1' in means that data is written to the FIFO and '0' means that no data is written to the FIFO) , *rd_en* is the read enable port ('1' in means that data is read from the FIFO and '0' means that no data is read from the FIFO), *clk* is the clock input port (this signal is used for synchronous read and write operations; all the operations are positive edge sensitive), *srst* is the reset or clear bit port (used to clear the contents of the FIFO and set the internal counter to the initial state), *full* (Full set to '1' indicates that all the address locations in the FIFO are filled), *almost_full* (almost_full set to '1' indicates that all the address locations in the FIFO are filled except the last location), *empty* (empty set to '1' indicates that all the address locations in the FIFO are unfilled) and *prog_empty* (prog_empty set to '1' indicates that all the address locations in the FIFO are unfilled except the number of locations programmed in the prog_empty bit) comprise the FIFO status bit ports of the memory unit. How the FIFO opeartes in this Halftoning system is described as follows.

- When the DDR2 RAM of Figure 3.1 is filled with atleast two consecutive rows of the image to be halftoned, it gives a signal to the Input Image FIFO inside the FPGA to start accepting the data. The controller inside the FPGA detects the

signal and the write enable port in the Input Image FIFO is set to '1' filling the FIFO at every positive clock edge.

- When the Input Image FIFO reaches the end of the stack which means all the locations are filled with data, the *full* and *almost_full* bits are set to '1' indicating to the controller that the FIFO is ready for operation.
- The processor core starts to read the data from the Input Image FIFO by setting the read enable port bit to '1'. When the FIFO reaches the end point where it needs to again fill up with data, the *empty* and *prog_empty* ports are set to '1' indicating the FIFO is empty.
- Data can be read and written to the Input Image FIFO simultaneously providing full duplex capability preventing latency. Both read and write operations are performed at the positive edge of the clock and have a latency of one clock cycle. The Input Image FIFO is run at 50 MHz, the same frequency as the entire system.

4.5 **Parameter Registers and 8/12 Bit Convertor**

The Parameter Registers of Figure 3.1 are memory components designed to store the input parameters of the Halftoning system. There are two registers, each 32 bits wide, used to store five input parameters; namely, Rows, Columns, Channels, Levels and 8/12 bit select. Figure 4.4 shows the distribution of the parameters in the register where the bits 0 through 15 are used to store the number of columns in the image and bits 16 through 31 store the number of rows in the image.

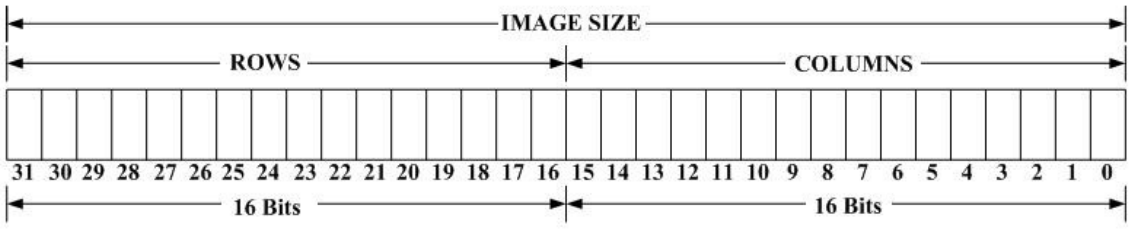

Figure 4.4: Parameter Register 1

Figure 4.5 shows the format for storing the information on the number of channels, number of levels per channel and the choice of 8 or 12 bit image input.

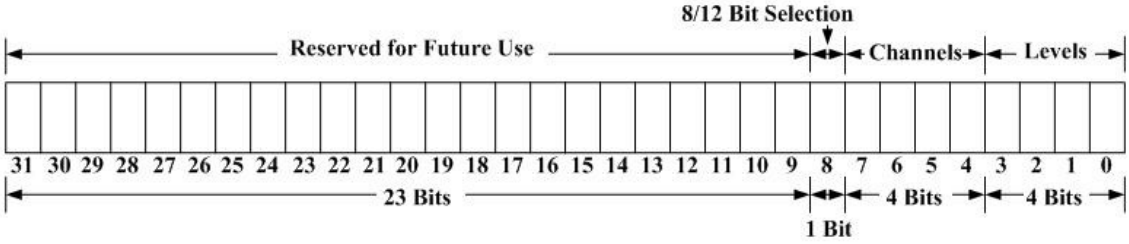

Figure 4.5: Parameter Register 2

Bits 0 through 3 are reserved for storing the number of levels per channel (color), bits 4 through 7 are used to represent the number of channels in the given image and the 8th bit in the format is used to switch between an 8 or 12 bit image input. A '0' in the 8th bit suggests that the input is 12 bits wide and a '1' indicates that the input is 8 bits wide. The remaining 23 bits are reserved for future use where additional parameters can be accommodated. This system design handles an image of width 24 inches and a resolution of 720 dpi. Thus, 24*720 gives the maximum number of columns accepted in this design which is equal to 17280. The height of the image, namely the number of rows in the image that this design can handle is infinite. For a specific system design a certain number is chosen for the sake of hardware power consumption. Thus, a image size of 44 inches which gives 31680 rows can be processed (Number of rows in the image can be of any size). If any of the parameters are not in the range mentioned, the hardware sends an error message to the host PC.

Figure 4.6 shows the design of the parameter register and this format is the same for any register in this hardware architecture. The parameters *rin[n:0]* is the input to the register where *n* is the Most significant bit of the data, *clr* is an input port used to clear the memory contents, *ls* is the load-store input control bit which performs load or store operations (when ls = '0', input is loaded in the memory register; when ls='1', input is stored in the register), *clk* is the clock input to synchronize the operations with the clock edge and *rout[n:0]* is the data output from the register.

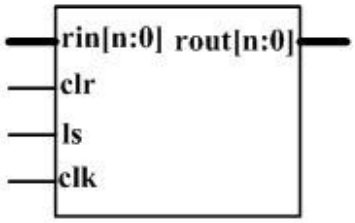

Figure 4.6: Register Schematic

Initially, during system start-up, the number of rows, columns, channels, levels and 8/12 bit select values are loaded/stored in the two parameter registers. The 8th bit in parameter register 2 is connected to the 8/12 bit convertor which is shown in Figure 4.7. The convertor converts a 8 bit data value into an equivalent 12 bit value by padding where the 8 bit value is left shifted four times and filled with '*1*' in the leftmost digits. The line number named '*2*' in Figure 4.3 shows the software conversion of the input pixel, thus this convertor is an equivalent hardware technique to change a 8 bit value to a 12 bit value. The output of the Input Image FIFO is connected to this 8/12 bit convertor as shown in Figure 3.1 and the circuit design of the convertor is shown in the Figure 4.8.

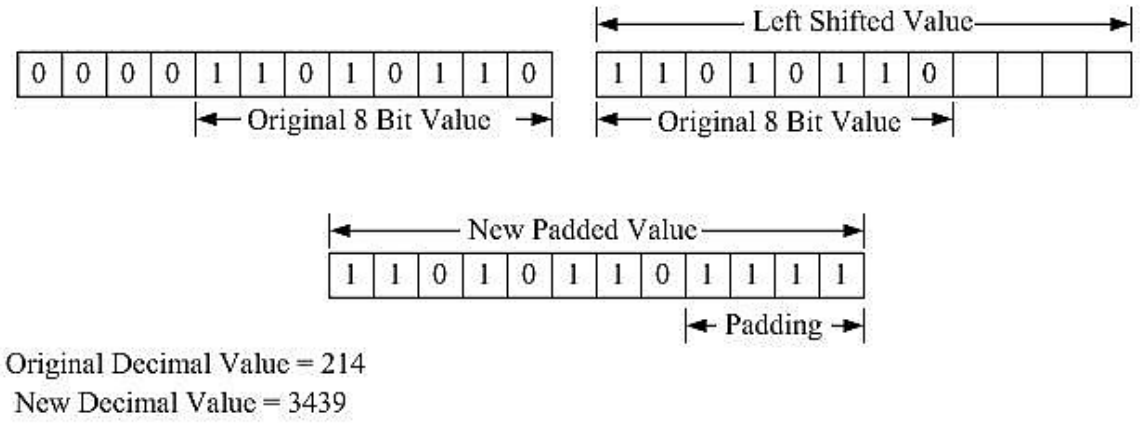

Figure 4.7: Padding Technique

The input port *d[11:0]* is the original value from the Image FIFO (it can be either 8 or 12 bit value), *s* is the select signal from the parameter register 2 that decides conversion ('0' in *s* suggests that the input value is a 8 bit value and hence the conversion takes place and

'1' in *s* indicates that the data value is a 12 bit value in which case the input its reflected at the output) and finally *dout[11:0]* is the output value (padded or un-padded depending upon the input).

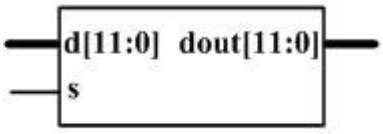

Figure 4.8: 8/12 Bit Hardware Convertor

4.6 **Droplet Densities Storage ROM**

This section deals with the input pixel being mapped to a maximum of three different droplet intensities with the help of a Single-Port ROM as shown in Figure 4.9. The output of the hardware convertor circuit is connected to the input of this ROM. As the input address is of width 12 bits, the number of locations addressed by the ROM is 4096 (2^{12}). The number of ROMs depend on the number of levels per channel which in this architecture is 1, 2 or 3 depending on the output requirement. Thus, a total of 3 Block RAM elements are required to store the droplet values. The values are calculated using the original algorithm and stored in the corresponding ROMs. The data in the ROM is 16 bit wide Fixed-Point format and integer arithmetic is performed. Hence, the input pixel value of 12 bits is mapped to three different droplet density values that are 16 bits wide. The amount of storage memory required to store these values is calculated as follows: 3 ROMs each with 4096 address locations supporting 16 bit data per location requires 3*4096*16 which is equal to 196608 bits of memory space that can be taken from the Block RAM column of the FPGA.

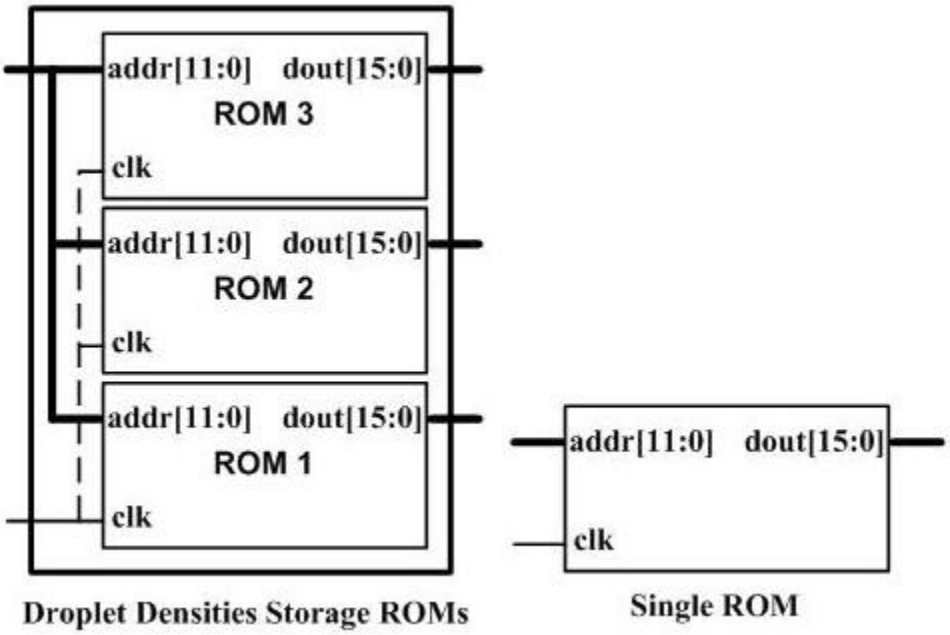

Figure 4.9: Droplet Densities Storage ROMs

The ROMs are generated using the Xilinx Core Generator wizard and the values are initialized using a *.COE* file in Xilinx as shown in Figure 4.10. The *.COE* file requires two parameters to be passed; one is the *memory_initialization_radix* and the other is the *memory_initialization_vector*. The *memory_initialization_radix* defines the format of representing the data which can be binary (radix = 2), octal (radix = 8) or hexadecimal (radix = 16). In this research the data format is binary. Figure 4.9 shows the ROM schematic where *addr[11:0]* is the 12 bit input pixel value, *clk* is the clock signal for synchronous operation (the input is positive edge dependent) and *dout[15:0]* is the output value related to droplet density. All the levels are grouped into a single ROM having the same input values but provides 3 different output intensities. This unit is also run at the system frequency of 50MHz.

```
memory_initialization_radix = 2;
memory_initialization_vector = 0000000000000000,0000000000000000,.....,0100000000000000,0100000000000000;
```

Figure 4.10: .COE File Format

4.7 **Input Level FIFO**

The Input Level FIFO is a digital memory component that acts as an intermediate buffer to store the values obtained from the Storage ROM described in Section 4.6 . Figure 4.11 shows the FIFO schematic where *data_in[15:0]* is the 16 bit wide input data, *clr* is the clear bit to reset the memory contents of the FIFO to a known value generally '0', *rd_en* is the control bit for read operation ('0' indicates no operation and '1' indiates read operation), *wr_en* is the write enable control bit for write operation ('0' indicates no operation and '1' indiates write operation), *clk* is the clock input port for synchronous operation (data is buffered with respect to the positive clock edge), *data_out[15:0]* is the 16 bit wide output port, *AFULL* is an output port which indicates to the control unit that all the memory locations except the last is filled, *AEMPTY* is a status bit that provides information that none of the locations except one is full and *FULL* is the output status bit that says that all the locations in the FIFO are filled. The current FIFO has a depth of 16 bits and stores 16 bit wide data.

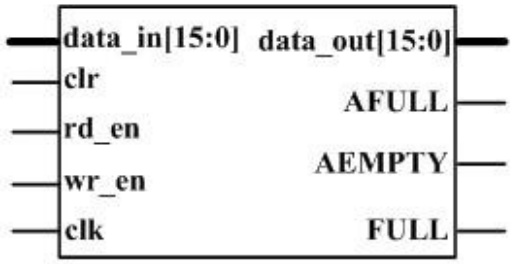

Figure 4.11: Input Level RAM/FIFO

As discussed in Section 4.2.2 , this FIFO design uses the Distributed RAM to achieve highest performance and minimum latency. Data can be read from or written to the FIFO simultaneously and maximum care is taken in the design to avoid read/write conflicts. The output from the Droplet Densities Storage ROM is connected to the 3 Input Level FIFO's. The amount of distributed RAM space consumed by these FIFO elements is 3*16*16 that shows 768 bits of storage. These FIFOs eliminate the latency that exists between the processor core and the Storage ROMs which would otherwise be 2 clock cycles. Their operation is similar to that of the Input Image RAM/FIFO shown in Section

4.4.2 but the only difference is that the Input Image FIFO uses a Block RAM and the Input Level RAM uses a Distributed RAM. This memory component is run at the system frequency of 50 MHz.

4.8 **Core Data FIFO**

The Core Data FIFO memory system acts as a cache to the Processor Cores where the data stored in the Input Level FIFOs is in turn buffered to the Core Data FIFOs. The current hardware architecture supports a maximum of 4 channels and 3 levels per channel which requires 12 Processor Cores. Thus, for 12 Processing Elements , 12 caches (Core Data FIFO) are required to buffer the correct input data to the designated core. The data from the Input Level FIFO is buffered and stored in the Core Data FIFO with the help of the control unit.

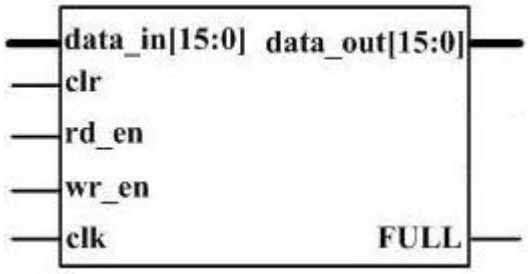

Figure 4.12: Core Data FIFO Schematic

Figure 4.12 shows the schematic of the Core Data FIFO which is similar to the Input Level FIFO except for the absence of *AFULL* and *AEMPTY* status ports. The design and operation of the Core Data FIFO is the same as the Input Level FIFO. The Core Data FIFO supports a data width of 16 bits and has a maximum depth of 4 locations. The memory required to implement 12 Core Data FIFOs is 12*4*16 which is equal to 768 bits. Distributed RAM is used to design this FIFO because the processing elements communicate with this FIFO at the highest speed and lowest latency. This component is also run at the system frequency of 50 MHz and the operations are performed at the positive edge of the clock. The output of the Core Data FIFO is connected to the registers in the Processor Cores of Figure 3.1 for subsequent computation.

4.9 Entire Input Data Memory Architecture

Figure 4.13 shows the fully connected memory elements of the entire Input Data Memory Architecture starting from the Input Image FIFO, Parameter Registers, 8/12 Bit Convertor, Droplet Densities Storage ROM, Input Level FIFO and Core Data FIFO. The Core Data FIFOs are arranged in a sequential order starting from channel 1 (that includes level 1, 2 and 3) and ending at channel 4 (includes level 1, 2 and 3). Initially the input parameters are loaded into the Parameter Registers. The data from the DDR2 SDRAM is buffered through the memory interface and is stored in the Input Image FIFO. Once the Input Image FIFO is completely filled with the input pixel data, the control unit starts the process of buffering the pixels to the processor core as follows. The data output from the Input Image FIFO goes through a 8/12 bit convertor circuit that converts a 8 bit value to the corresponding 12 bit value and feeds it to the Droplet Densities Storage ROM where the input pixel is mapped to three different ink intensities. The Droplet Densities Storage ROMs 1, 2 & 3 constitute a single unit named as Droplet Densities Storage ROM as shown in Figure 4.13. During this time the Input Image FIFO is constantly monitored by the control unit to fill the FIFO in case it is empty. All the operations are executed in parallel thus improving the execution speed of the algorithm. The output from the Storage ROM is stored in the Input Level FIFO where it is finally buffered to the Core Data FIFO. The Core Data FIFO is arranged in such a way that if the image has only one channel, 3 levels per channel, then Channel 1 is chosen. If there are 2 colors in the input image then Channels 1 and 2 are selected, else if there are 3 colors, channel 1,2 and 3 are selected else of there are 4 colors, channels 1,2,3 and 4 are selected. All the memory components are positive edge sensitive and run at the system frequency of 50 MHz. The memory unit is designed such that when the Processor Cores start processing the input data, the architecture ensures that the data is continuously buffered from the input to the Processing Elements. Thus, from the Processor Core point of view, the data from the input data memory is obtained every clock cycle without any interruption but actually it takes 5 clock cycles to reach the Processor registers. The memory buffering devices like Input Image FIFO, Input Level FIFOs and Core Data FIFOs are automatically filled.

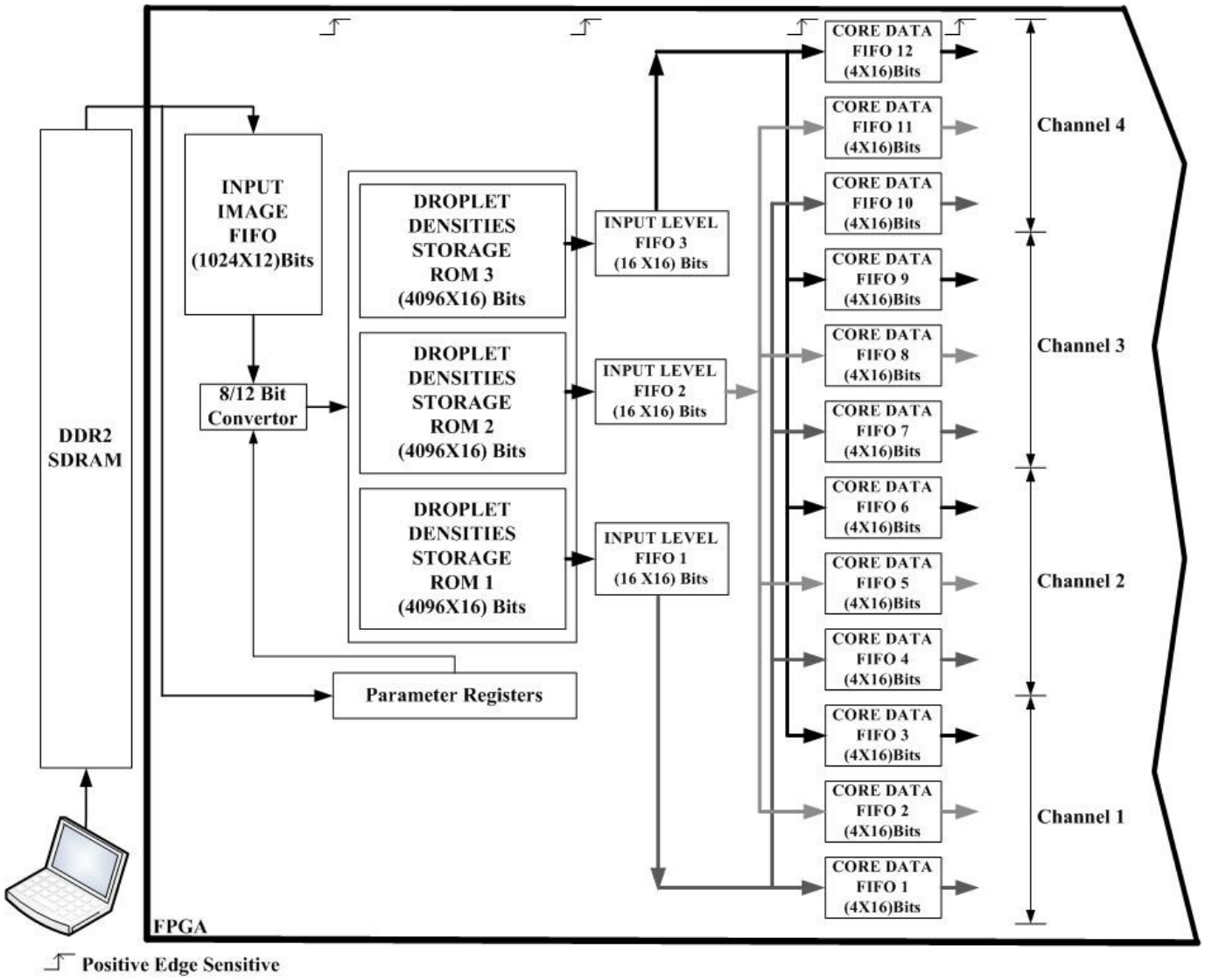

Figure 4.13: Entire Input Data Memory Architecture

Chapter 5. Processor Core Architecture Development and Design

5.1 Introduction

This chapter gives a thorough insight into the development and design of the Processor Core architecture for the Processor Cores of Figure 3.1. The Processor Core elements are the most critical part of the hardware system in that they perform all the arithmetic, logical and memory operations. This architecture uses Xtreme DSP slices for addition, subtraction and multiplication as these are hard cores embedded inside the FPGA to achieve the highest performance. The main components that constitute the Processor Core are Input Data Registers, a Adder-Subtractor, Threshold Comparators, Error Limiting Circuit, Error Registers, Random Weights/Values Generators and Error Filter Circuit that consists of Multipliers and Adders. The halftoning algorithm written in 'C' is shown in the code snippets of this chapter and the equivalent implementing hardware unit design is designed.

5.2 Xilinx Virtex-5 Xtreme DSP Slice

The Xtreme DSP [28] slice in a Virtex-5 FPGA chip is also called DSP48E and it is used for high speed digital signal processing. The DSP48E is a hard element which is etched into the FPGA chip as shown in the Figure 5.1.

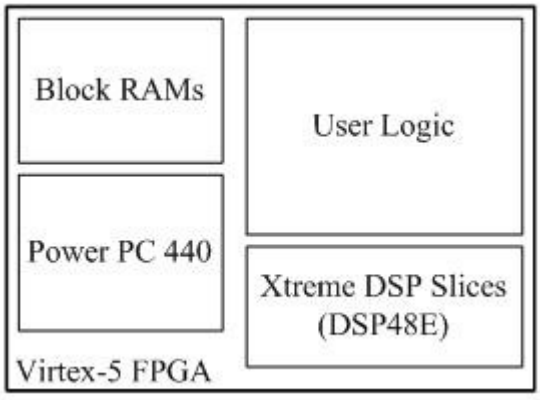

Figure 5.1: Virtex-5 FPGA Components

These slices can be used for functions such as multiply, multiply-accumulate, multiply-add, add, subtract, barrel shift, bit wise logical operations, magnitude comparator and counter. The merits of using the DSP48E components is because of the flexibility, improved efficiency, reduced overall power consumption, increased maximum frequency and reduced set-up plus clock-to-out time. The Processor Core architecture uses a lot of these components to reduce the FPGA device utilization as the adder, subtractor and multiplier is already available embedded in the FPGA , there is no need to design or develop a new adder or multiplier circuit which takes substantial device resources and a lot of time to test. The hard IP cores are fully tested and there is a faster execution of operations that results in increased performance which is crucial for the Processor Core. Another advantage of using the DSP components is that more functionality can be added to the user design as the arithmetic unit utilizes 0% of the device, thus a bigger system can be implemented on the FPGA. The unit supports both signed and unsigned data arithmetic where it uses 2's complement methodology. The maximum frequency at which the slices can be run is 550 MHz when a fully pipelined architecture is used. To support higher data widths, the DSP slices can be cascaded without any downfall in the performance. Using the DSP slices decreases the design and verification time of the hardware architecture developed as a newly fabricated design would take a considerable amount of time for verification and validation. The DSP slices are instantiated into the hardware design with the help of the Xilinx Core Generator wizard. In this research, the DSP48E slices are used to perform signed addition, subtraction and multiplication. The number of DSP slices in the XC5VFX70T FPGA is 128 and the current hardware architecture uses 108 of these slices. As discussed in Chapter 3., there are 12 Processor Cores and each Core consumes 9 DSP48E slices and hence the total comes to 108. Absolutely, no FPGA User Logic resources are utilized when these slices are instantiated into the design. On the other hand, if the DSP slices were not used, there would have been a tremendous increase in user logic resources. Typically more than 5000 slices would have been required which results in a device utilization of over 80%. Thus, the DSP48E slices play a major role in this hardware system by reducing the amount of on-chip resources consumed, the design time is reduced by half and performance of the system is

increased. The following sections give in-depth details about the Processor Core Architecture.

5.3 Input Data Registers

The input data registers are used to store the input data to the arithmetic unit for a specified amount of time before the next data comes into the processing unit. There are two 16 bit input registers namely *Input Pixel Register* and *Previous Pixel Register*. The *Input Pixel Register* takes the data from the Core Data FIFO and consists of values of each color component of the original image. The *Previous Pixel Register* is used to store the error value diffused from the previously processed pixel location. The schematic of these two registers is similar to the parameter registers shown in Figure 4.6 except that the size of the register here is 16 bits. The format in which the data is stored in these registers is shown in Figure 2.3. Figure 5.2 shows the software code snippet used in the original halftoning algorithm written in 'C' that gives information about the input data operation and the equivalent hardware circuit is shown in Figure 5.3.

```
for (c=0; c<imageChn; c++)                                    C-1
{
  levels=numLevelsPerChannel[c];
  for (l=0; l<levels; l++)                                    C-2
  {
   index=j+n*imageLvl+(rowIndex%bufferSize)*imageLvl*imageCol;

   inputPixel=pixelBuffer[j+n*imageLvl];                      C-3

   outputImage[index]=inputPixel+errorImage[index];           C-4
```

Figure 5.2: Software Code Snippet For Registers and Adder

The following points are derived from the software code.

- '*C-3*' is a part of the software code where '*inputPixel*' represents the Input pixel Register in hardware. The term '*pixelBuffer[c]*' is the software buffer created to provide the input pixel data to the inputPixel. Where the term '*c*' represents the

color component of a particular pixel. The variable '*errorImage*' is the buffer created in the 'C' code to store the previously processed pixel value.

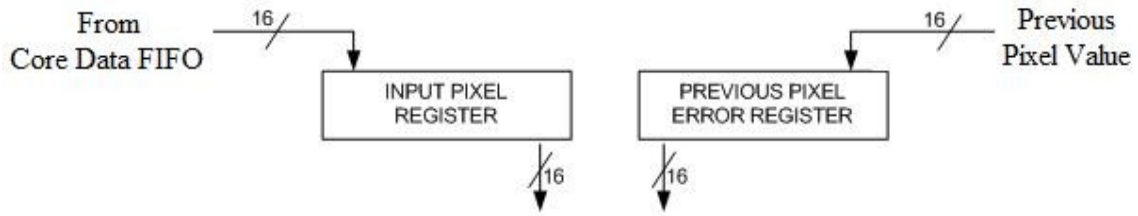

Figure 5.3: Equivalent Hardware Circuit for Input and Previous Pixel Values

- The equivalent hardware circuit in Figure 5.3 shows that the Input Pixel Register (*'inputPixel'*) is connected to the Core Data FIFO (*'pixelBuffer[c]'*) and the Previous Pixel Register is equivalent to the '*errorImage[e]*' where '*e*' is the error from the same component (same level in a channel) of the previous pixel. The registers are positive edge triggered and takes one clock cycle to store the data.

5.4 **Adder-Subtractor Unit**

This unit performs the addition/subtraction operation depending on a control bit. Xilinx Core Generator is used to create the Adder-Subtractor unit which uses a DSP48E slice. The adder schematic is shown in Figure 5.4 where *'AB_IN*', '*C_IN*' are the input ports (signed) to which the input value registers are connected, '*CE_IN*' is the clock-enable port that controls the operations ('0' means device is inactive, '1' means that the device is active for operation), it is synchronized with the positive edge of the clock and has the highest priority over other signals, '*RST_IN*' is the clear or reset bit that sets the output of the unit to zero ('1' in this port drives the output to zero and '0' in this port means a normal device operation; this port is normally used during system start-up), '*SUBTRACT_IN*' is the control input port that decides the type of operation to be performed ('0' means Add operation and '1' means Subtract operation), '*CLK_IN*' is the clock port for connecting the clock signal resulting in a synchronous operation and '*P_OUT*' is the output port that provides the result of the operation used. There is no need for an overflow indicator in this unit as the values strictly lie between -1 and 1.

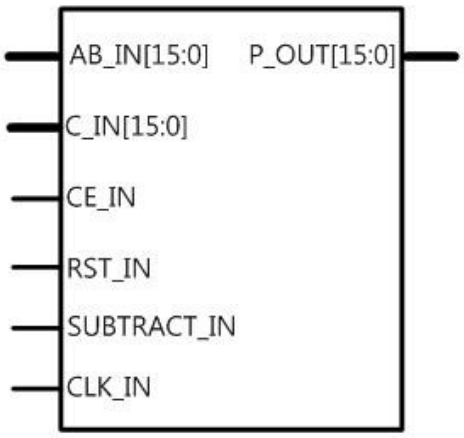

Figure 5.4: Adder-Subtrator Unit

Schematic

The code snippet in Figure 5.2 delivers the following information.

- '*outputImage[c]*' shown in *'C-4*' is a buffer location where the sum of '*inputPixel*' and '*errorImage[e]*' is stored.

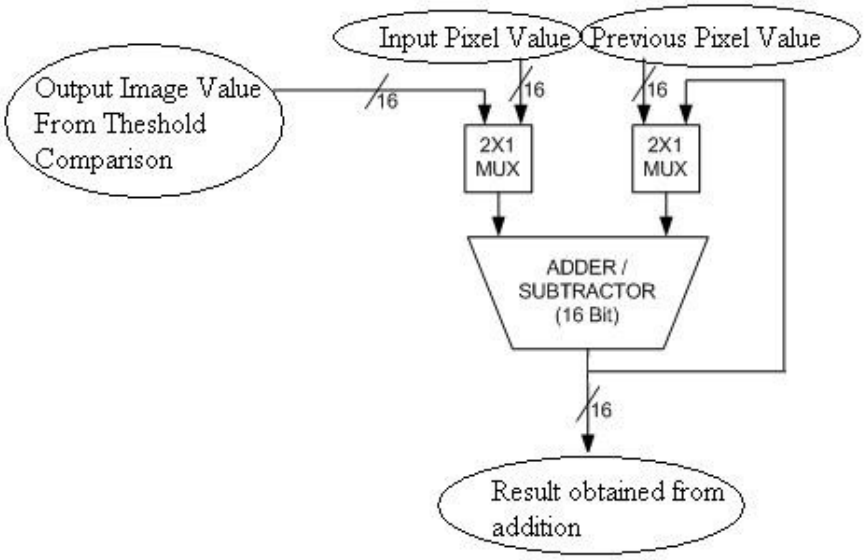

Figure 5.5: Adder-Subtractor Connections

- The Adder-Subtractor component shown in Figure 5.5 shows that each of the input ports is connected to two different data inputs with the help of a multiplexer as both addition and subtraction has to be performed. The '*outputImage[c]*' in the

software is equivalent to the result obtained at the output port of this hardware unit. The hardware unit is positive edge triggered and takes one clock cycle to output the result.

5.5 Threshold Comparison Circuit

The threshold comparison circuit compares the output of the Adder-Subtractor unit with a constant (0.5 in this research) and produces an output depending on the comparison. The software code snippet for this operation is shown in Figure 5.6. The following can be inferred from the software code.

```
if (l==0) // no stacking constraint                                    C-5
{
  if (outputImage[index] >= Threshold)
  {
     outputImage[index]=1.0;
  }
  else
  {
     outputImage[index]=0.0;
  }
}
else // enforce stacking constraint                                    C-6
{
  if ((pixelBuffer[j+n*imageLvl-1] > 0.5) && (outputImage[index] >= Threshold))
  {
     outputImage[index]=1.0;
  }
  else
  {
   outputImage[index]=0.0;
  }
}
```

Figure 5.6: Software Code Snippet for Threshold Comparison

- The code in '*C-5*' indicates that it is applicable only for the first level (droplet density) in any channel. It compares the '*outputImage[c]*' which is the result from the Adder-Subtractor unit with the threshold value (0.5), if the result is greater

than or equal to the threshold then '*outputImage[c]*' is replaced by the value 1.0 and if less than 0.5 '*outputImage[c]*' is replaced by the value 0.0.

- The code in '*C-6*' is applicable to any level except the first, the codes '*C-5*' and '*C-6*' are similar but the only difference is the term '*pixelBuffer[c-1]*' which is the output value of the previous level (either 1.0 or 0.0) in the same channel in the same pixel.

The design of equivalent hardware for the threshold comparator circuit is described below.

- The Threshold Comparison circuit consists of three main components namely Threshold Comparator, Previous Value Register and Output Image Value Circuit.
- The Threshold Comparator is generated with the help of the Core Generator wizard and it performs signed comparison with a constant threshold value of 0.5. The Figure 5.7 shows the schematic of the comparator where '*a[15:0]*' is the input port connected to the result of the Adder-Subtractor unit, '*Constant 0.5*' is the value with which the result is to be compared and '*a_ge_b*' is the output from the comparator.

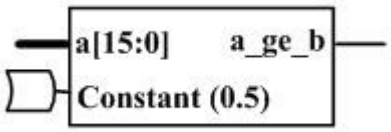

Figure 5.7: Threshold Comparator

- The next important circuit is the Output Image Value Circuit shown in Figure 5.8 where '*cmp*' is the input port connected to the output of the threshold comparator, '*pv*' is the previous core value (previous level), '*en*' is used to control the circuit ('0' in '*en*' means no operation and '1' means regular operation), '*ov[15:0]*' is the 16 bit output value and '*nv*' is the value to the next processing element (typically next level in the same channel). This circuit gives an output image value of 1.0 if all the three inputs are 1 and 0.0 otherwise.

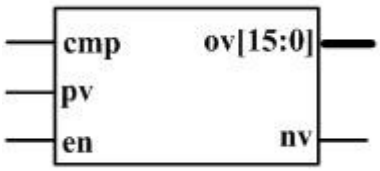

Figure 5.8: Output Image

Value Circuit

- Figure 5.9 shows the Threshold Comparison Circuit where there is only one hardware unit for all the levels in a channel unlike the two code segments shown in Figure 5.6. The difference is evident from the Previous Core Value register which is set to the value '1' for the first level in all the channels. This is done by driving the reset bit in the register to '1'. For all other cores that doesn't represent the initial level, the reset bit of the previous value register bit is disabled (tied to '0'). This reduces the unnecessary replication of hardware circuits.

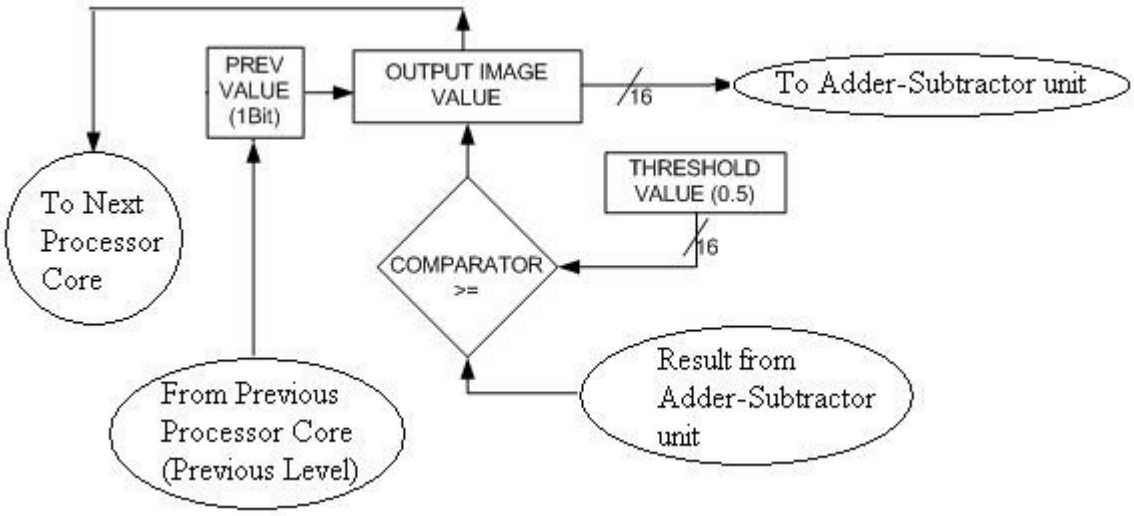

Figure 5.9: Threshold Comparison Circuit

- As the output from the Threshold Comparison Circuit is ceiled (rounded to the nearest integer value) to one of the two values (1.0 or 0.0), there arises an error which is the difference between the original value and the ceiled value. The Figure 5.10 shows the software code snippet '*C-7*' where the output value is subtracted from the result of the addition giving the error value. The Adder-Subtrator unit is

used for this subtraction operation with the help of two 16 bit multiplexers controlled accordingly.

```
errorImage[index]=errorImage[index]+inputPixel-outputImage[index]; C-7

if (errorImage[index]>ErrorLimit)
errorImage[index]=ErrorLimit;                                      C-8
else if (errorImage[index]<-ErrorLimit)
errorImage[index]=-ErrorLimit;
```

Figure 5.10: Code Snippet for Subtractor and Error Limiting Circuit

5.6 **Error Limiting Circuit**

This circuit is implemented to monitor the error value that accumulates over a period of time when the pixels are being processed. The error value is constrained to a range of values strictly between -1 and 1. The code snippet '*C-8*' shown in Figure 5.10 infers that when the error value is greater than the '*ErrorLimit*' (value is 1), then the value of '*errorImage[e]*' is set to '1', but when the error value is less than '-*ErrorLimit*' (negative of '*ErrorLimit*' which is '-1') the value of '*errorImage[e]*' is set to '-1' else if neither of the conditions is satisfied, the error value remains unchanged. The equivalent hardware conversion is shown in Figure 5.11.

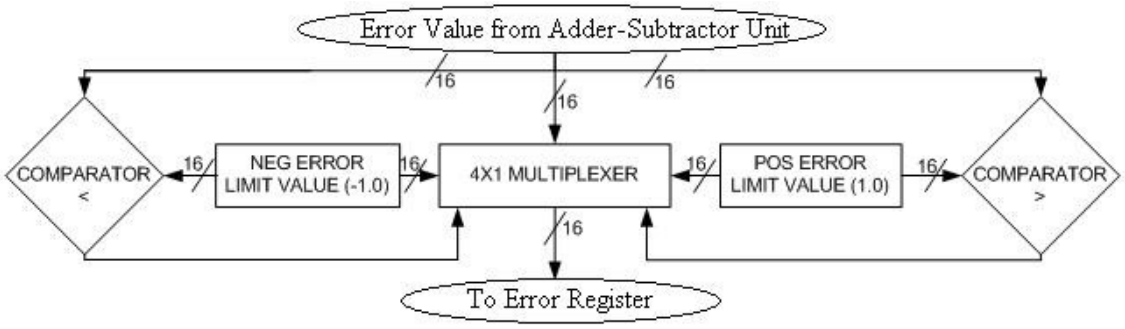

Figure 5.11: Error Limiting Circuit

Both the Comparators are IP cores generated by using Core Generator software which has a 'greater than' function and the other with a 'less than' function. The '*Neg Error Limit Value (-1.0)*' and '*Pos Error Limit Value (1.0)*' are combinational circuits producing a

constant value of -1 or 1. The multiplexer is used to select one of the values based on the output from the comparators. Figure 5.12 shows the schematic of both the comparators where '*a_gt_b*' and '*a_lt_b*' are output bits obtained after comparison ('0' when the condition is false and '1' when the condition is true).

Figure 5.12: Comparators (Greater Than and Less Than)

5.7 **Error Registers**

There are two Error Registers present in the Processor Core, one for storing the error value output from the Error Limiting Circuit and the other to store the previously stored error values in the Error Storage Block RAMs. The registers have the same design as the Input Value registers that are 16 bits wide shown in Figure 5.13.

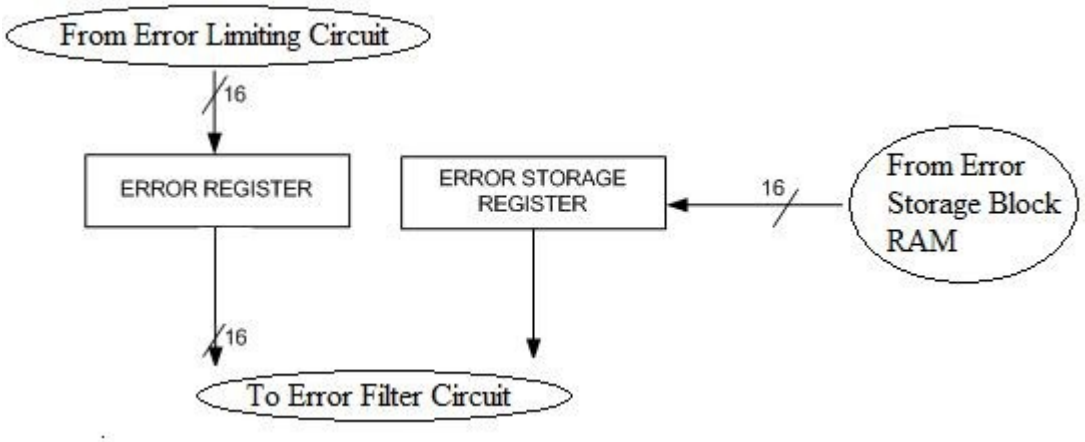

Figure 5.13: Error Registers

The code snippet '*C-8'* in Figure 5.10 indicates '*errorImage[e]*' which is equivalent to the Error Register hardware unit. The registers are positive edge sensitive digital elements and are very fast compared to Distributed or Block RAMs. They provide synchronous operation avoiding timing conflicts. The upper half of this unit deals mainly with calculating the output and the error value and the lower half gives details about the error diffusion circuit with random weights generator circuits.

5.8 **Random Weights-Values Generator**

The importance and use of randomness in an image is discussed in Chapter 1., Section 1.1.5 . The software code snippet in Figure 5.14 shows the random numbers generated using 'C' program. Thus, to design an equivalent hardware random number generator, the 'C' code must be thoroughly analyzed in order to match the outputs with the software unit.

```
void LAUsetWeights(float *weights)
{
  float frand;
  weights[1]=0.0;
  weights[2]=0.0;

  frand=((float)rand()/(float)RAND_MAX-0.5)*5.0/16.0;      C-9
  weights[0]=7.0/16.0-frand;
  weights[4]=5.0/16.0+frand;

  frand=((float)rand()/(float)RAND_MAX-0.5)*1.0/16.0;      C-10
  weights[3]=3.0/16.0-frand;
  weights[5]=1.0/16.0+frand;

  return;
}
```

Figure 5.14: Code Snippet for Random Weights Generation in 'C'

To design a hardware unit to generate the random weights and numbers, code snippets '*C-9*' and '*C-10*' in Figure 5.14 have to be fully broken down into small segments for easier understanding of the number generation process. All the floating-point numbers are converted to equivalent Fixed-Point numbers and the range of numbers for '*frand1*' shown in the code snippet is determined. The calculated range of values for '*frand2*' in code '*C-9*' is [-0.15625,0.15625] (Floating-Point), [-2560,2560] (Fixed-Point) and in code '*C-10*' is [-0.03125,0.03125] (Floating-Point), [-512,512] (Fixed-Point). The fixed-Point number for the weight [7/16] is 7168, for [5/16] it is 5120, for [3/16] it is 3072 and for [1/16] it is 1024. Thus the added sum of all the weights should be equal to 1 or 16384 (Fixed-Point). The value range for all the weights are shown in the following Equations 5.1, 5.2, 5.3 and 5.4.

$$weight[0] = \frac{7}{16} - frand1 = 7168 - ([-2560 \text{ to } 2560]) \quad (5.1)$$

$$weight[4] = \frac{5}{16} + frand1 = 5120 + ([-2560 \text{ to } 2560]) \quad (5.2)$$

$$weight[3] = \frac{3}{16} - frand2 = 3072 - ([-512 \text{ to } 512]) \quad (5.3)$$

$$weight[5] = \frac{1}{16} + frand2 = 1024 - ([-512 \text{ to } 512]) \quad (5.4)$$

Random numbers can be generated in hardware with the help of a Linear Feedback Shift Register (LFSR) circuit [29], [30]. A LFSR is a sequential digital circuit designed with the help of D-Flip Flops and XOR gates. The design depends on the basic fundamentals of polynomial arithmetic in cyclic coding theory. If there are '*n*' binary bits, the LFSR produces a sequence of $(2^n - 1)$ different non-zero values. The circuit needs to be designed in such a way that it satisfies the generating function called '*Primitive Polynomial*'. A '*Primitive Polynomial*' is an irreducible polynomial that produces all the elements in a given set. The sequence of random numbers generated by the LFSR has a property in which a number will never be repeated until the whole sequence of numbers are executed. The LFSR acts as a counter except that it produces randomness without incrementing the result by 1 and it is faster than any other counter. Any digital counter can count a particular range of values depending on the number of binary bits. Thus, for representing the '*frand1*' value of range [-2560,2560], the number of bits used must be adjusted. The nearest number range that matches the above range is [-2048,2047] which requires 12 binary bits to represent the whole range. '*frand2*' values range from [-512,512] which can be represented exactly by 10 bits except the value 512. Hence from this discussion it can be concluded that apporximately 95% of the random values can be generated similar to the software 'C' code. This doesn't affect the performance or the quality of the image as there is enough randomness introduced into the system to avoid the artifacts. As the number of binary bits required are known, a primitive polynomial has to be found that produces all the values in the range supported by the number of bits. The polynomial is defined by the powers of '*x*' as shown in the Equations 5.5 and 5.6.

$$Primitive\ polynomial\ (10\ \text{bits}) = x^{10}+x^{3}+1 \qquad (5.5)$$

$$Primitive\ polynomial\ (12\ \text{bits}) = x^{12}+x^{7}+x^{4}+x^{3}+1 \qquad (5.6)$$

LFSR consists of only D-Flip Flops and XOR gates. The gates are placed according to the primitive polynomial. One end of the register is always 1 ('x^0') and the other end is always 'x^n' where 'n' is the number of bits used to generate the random numbers. Figure 5.15, 5.16 shows the design of LFSRs for 10 and 12 bits respectively.

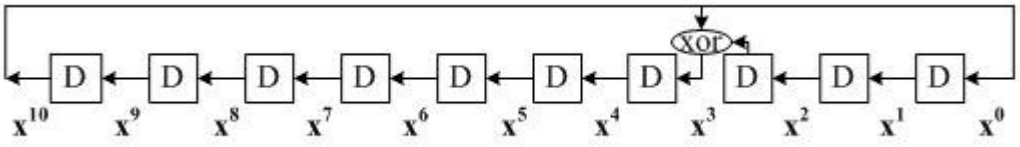

Figure 5.15: LFSR - 10 Binary Bits

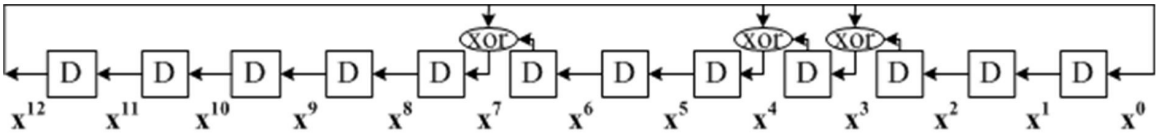

Figure 5.16: LFSR - 12 Binary Bits

To generate the weights [0,4,3,5], the random number generator hardware must interface with an adder logic as from Equations 5.1, 5.2, 5.3 and 5.4, the '*frand1*' or '*frand2*' is added to the weight filter [7/16,5/16] and [3/16,1/16]. A two's complement Adder/Subtractor is designed with one port of the adder tied to a constant value and the other port is connected to the random number generator. The random weight filter is divided into two units one with '*frand1*' and the other with '*frand2*' as constant values. Each random weight filter has two adder/subtractor units, each connected to one of the four constant values shown in the Equations 5.1, 5.2, 5.3 and 5.4. Both hardware units are combined to form a fully functional random weights generator unit. Figure 5.17 shows the random weights generator hardware unit where '*clk*' represents the synchronous clock, '*rst*' is the clear bit that resets the registers to the initial setting, '*en*' is the control port that controls all the operations ('0' – no operation, '1' – normal operation), '*wts0*' is the random weight[0], *'wts4*' is random weight[4], '*wts3*' is the random weight[3] and '*wts5*' is the

random weight[5]. This hardware unit is a positive edge sensitive circuit. The output from this unit is connected to the Error Filter Circuit.

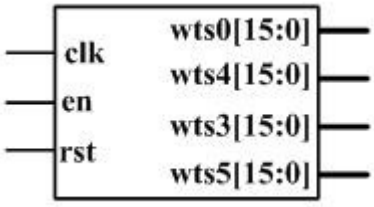

Figure 5.17: Random Weights Generator

5.9 Error-Filter Circuit

```
m_p=rowIndex+1;                                                          C-11
n_p=n;
if (m_p >= 0 && m_p < imageRow && n_p >= 0 && n_p < imageCol)
  {
    indexPrime=j+n_p*imageLvl+(m_p%bufferSize)*imageLvl*imageCol;
    errorImage[indexPrime] += errorImage[index] * weightMatrix[4];        C-12
  }

m_p=rowIndex;
n_p=n+d;
if (m_p >= 0 && m_p < imageRow && n_p >= 0 && n_p < imageCol)
  {
    indexPrime=j+n_p*imageLvl+(m_p%bufferSize)*imageLvl*imageCol;
    errorImage[indexPrime] += errorImage[index] * weightMatrix[0];        C-13
  }

m_p=rowIndex+1;
n_p=n-d;
if (m_p >= 0 && m_p < imageRow && n_p >= 0 && n_p < imageCol)
  {
    indexPrime=j+n_p*imageLvl+(m_p%bufferSize)*imageLvl*imageCol;
    errorImage[indexPrime] += errorImage[index] * weightMatrix[3];        C-14
  }

m_p=rowIndex+1;
n_p=n+d;
if (m_p >= 0 && m_p < imageRow && n_p >= 0 && n_p < imageCol)
  {
    indexPrime=j+n_p*imageLvl+(m_p%bufferSize)*imageLvl*imageCol;
    errorImage[indexPrime] += errorImage[index] * weightMatrix[5];        C-15
  }
```

Figure 5.18: Code Snippet for Error Filter Circuit

The hardware circuit discussed in this section is one of the critical units used to diffuse the errors generated at each pixel location. Figure 5.18 shows the software

implementation of the error filter circuit in 'C' code. The code in '*C-11*' shown in Figure 5.18 describes the ON-OFF timing locations in the image and is also mentioned in Figure 3.4. Code '*C-12*', '*C-13*', '*C-14*' and '*C-15*' are the four error filters used to diffuse errors to the neighboring pixels. Figure 5.19 shows how the error value at a particular location is updated. Let '*P1*', '*P2*', '*P3*', '*P4*', '*P5*', '*P6*' be the pixels of an image and '*EP1*', '*EP2*', '*EP3*' be the errors at the corresponding pixels '*P4*', '*P5*', '*P6*', the error '*EP1*' cannot be used for processing until all the pixels '*P1*', '*P2*', '*P3*' are processed. The code snippets '*C-12*', '*C-13*', '*C-14*' and '*C-15*' shows two important variables '*errorImage[c]*' and '*weightMatrix[w]*' that are multiplied and added to the previous error value at corresponding locations.

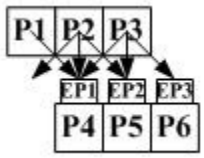

Figure 5.19:
Error Update
Technique

The hardware equivalent circuit uses multipliers, adders and register components to efficiently perform the error diffusion mechanism. The multipliers and adders in this circuit are implemented using the Xilinx Core Generator. Both the units are performed signed operations and the multiplier unit has an output port the same width as the input port. The implementation of the multiplier is shown in Figure 5.20 where '*a[15:0]*', '*b[15:0]*' are the input ports connected to the error register and the random weight generator, *'ce*' is the clock-enable pin with highest priority that controls the multiply operation ('0' – no operation, '1' – normal operation), '*sclr*' is the clear bit used to reset the multiplier output to a known value done during system start-up, '*clk*' is the clock input as the unit is synchronized with a clock and finally '*p[29:14]*' is the truncated output of the multiplier circuit. The process of truncation doesn't affect the output and as the data bus is 16 bits wide, the result needs to be broken down taking the useful value alone. The adder circuit is similar to the Adder-Subtractor circuit discussed earlier in this chapter in

Section 5.4 , the only difference being the absence of the '*SUBTRACT_IN*' input port as the subtraction operation is not necessary. From Code snippets '*C-12*', '*C-13*', '*C-14*' and '*C-15*', it is evident that four multipliers, adders and registers are required to handle the errors. The reason for using four 16 bit wide registers can be inferred from the Figure 5.19 that the error needs to be stored and propagated among the register circuits till the error value is fully updated. Each error filter unit is arranged in a way that it diffuses the error value generated to corresponding pixel.

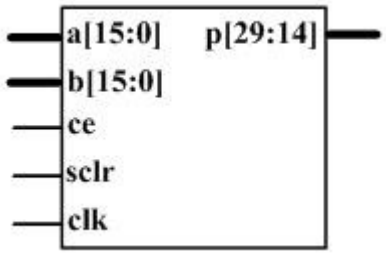

Figure 5.20: Multiplier Unit

The Error-Filter unit is shown in Figure 5.21 where there exists two random value generator circuits for randomizing the error values, Random Value Generator 1 generates values for all the input pixel locations but Random Value Generator 2 is enabled for certain pixel locations only. The error value from the error register is given as one of the inputs to the multiplier circuit and the random weight-values from the random weight generator is used as the second input. Each of the adder units is connected to a 16 bit register which stores the partial error value and shifts the value during successive pixel operation. The Error-Filter [7/16] gives the error value of the next unprocessed neighbor, thus the output of this filter is directly connected to the Previous Pixel Value Register. The final updated value of the error is stored in the Error Storage Block RAM. The stored error values are buffered through the Error Storage Register and finally to the error filter [7/16] to add the error at the particular pixel location. The equivalent hardware circuit for code '*C-12*' is the Error-Filter [5/16], '*C-13*' is error filter [7/16], '*C-14*' is the Error-Filter [3/16] and '*C-15*' is the Error-Filter [1/16] respectively.

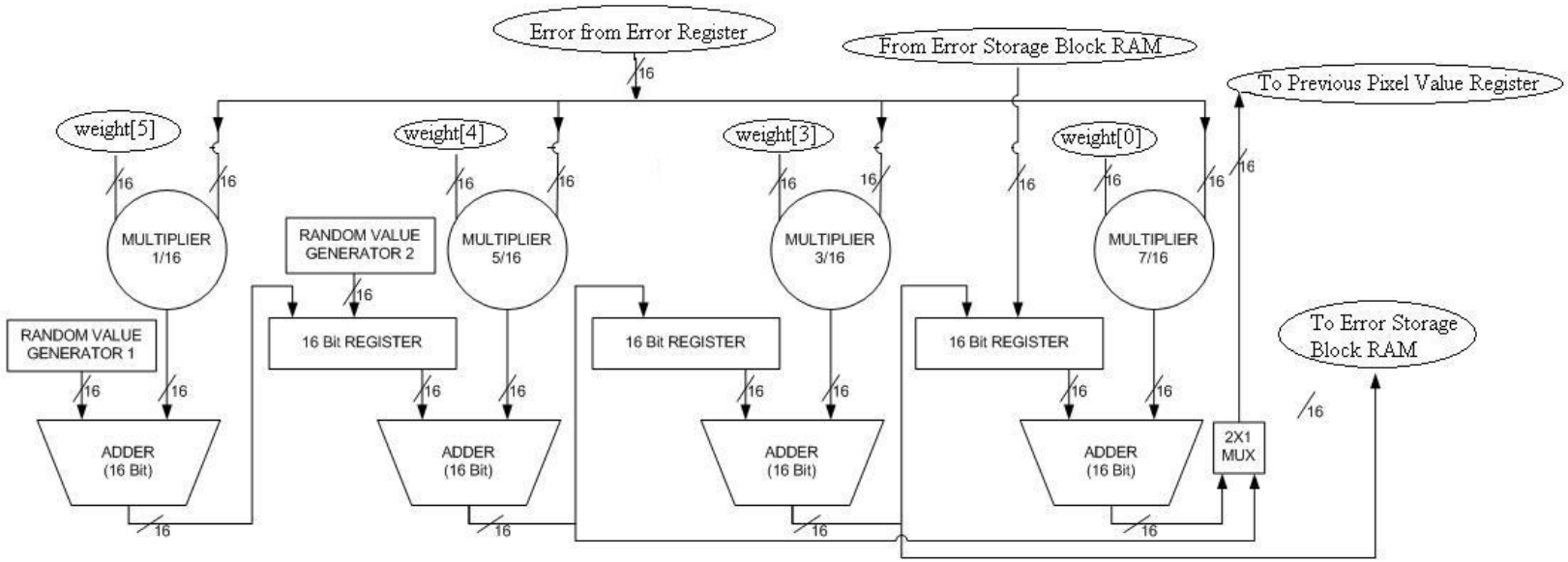

Figure 5.21: Hardware Error-Filter Circuit

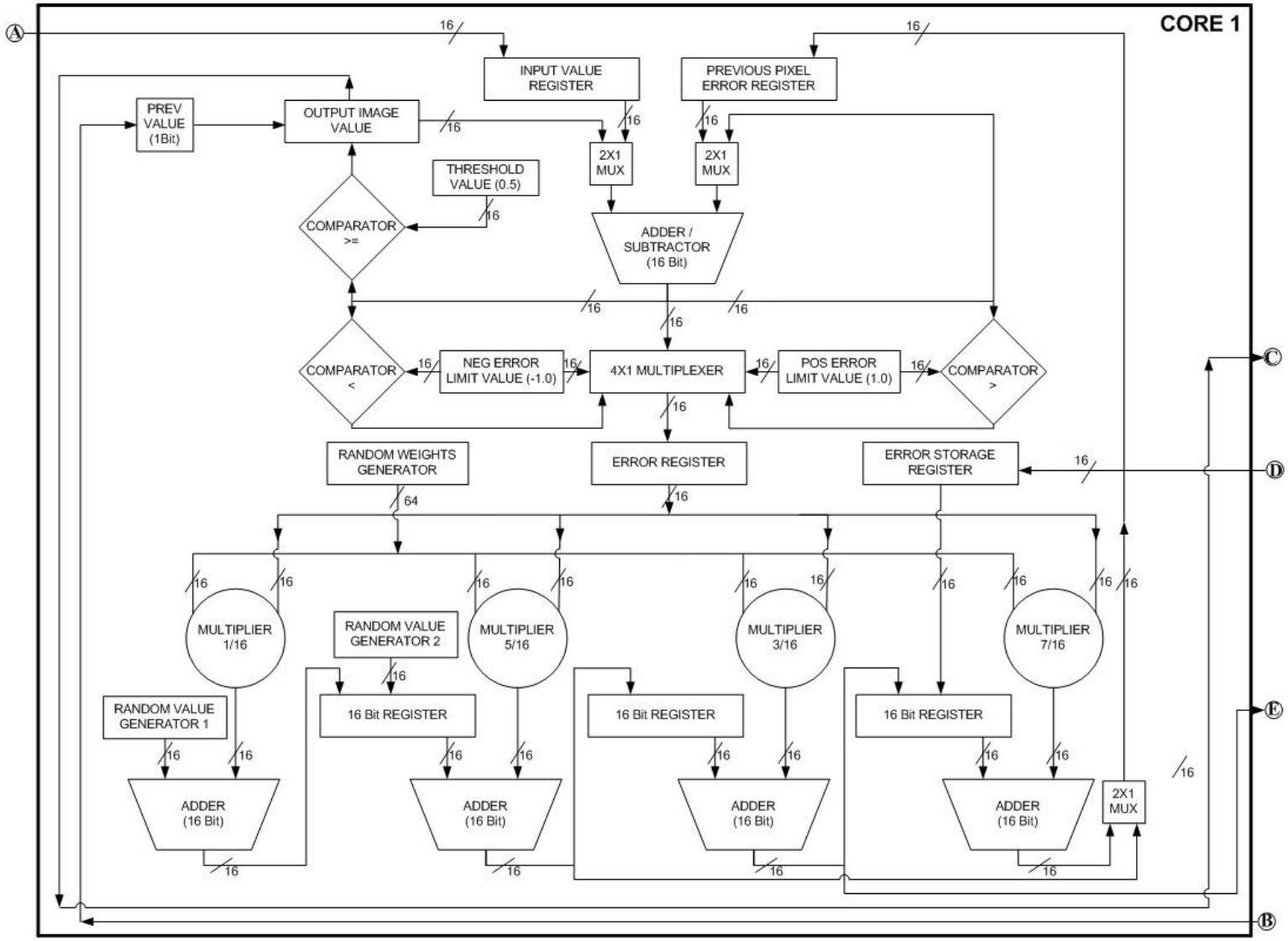

Figure 5.22: Processor Core Functional Architecture

5.10 **Processor Core Architecture**

Figure 5.22 shows the full schematic of a single Processor Core unit of Figure 3.1 where '*A*' is the data input pixels from the Core Data FIFO, '*B*' is the Previous Core (Level) value, '*C*' is the value from the present Core connected to the next Core, '*D*' is the stored error values from the Error Storage Block RAM and '*E*' is the final error value from the Error-Filter circuit connected to the Error Storage Block RAM for storage. Output from each Processor Core also represented as '*C*', is obtained every 8 clock cycles and the whole system runs at a frequency of 50 MHz which is also the system frequency.

Chapter 6. Error Storage Block Memory Architecture Design

6.1 Introduction

This chapter introduces the detailed concepts and information about the memory unit designed to store the errors generated by a Processor core of Figure 3.1 at every pixel location. This memory system is a most essential unit in the architecture and handles the error storage operations and is responsible for efficient operation of the Error-Filter Circuit in the Processor Core. All the hardware modules in this chapter are described in Verilog HDL and fully tested using the ModelSim simulation tool.

6.2 Error Storage Block RAM Architecture

The size of each Error Storage Block RAM depends on the image width and the data width of the values generated. The high level system architecture of Figure 3.1 consists of 12 cores, so the number of error storage blocks is equal to 12. The system supports an image width of 24 inches and a resolution of 720 dpi which gives 17280 pixels in a given row. The memory size doesn't depend on the number of rows or height of an image. The number of address locations in the given memory should be 17280 and each address space supports data of 16 bits. The total memory required for storing the errors generated by all the cores is 207376 (12*16*17280) bits. Since it requires a large memory to accommodate the data, Xilinx Block RAMs are used. In order to get the most efficient and reliable design, the memory system is designed using the Xilinx Core Generator wizard. The design uses a simple dual-port RAM configuration shown in Figure 4.1 where the data can be read from or written-to the memory simultaneously. Figure 6.1 shows a higher level schematic of the Error Storage Memory Block where '*dina[15:0]*' is the input port to the memory that transfers the error data, '*doutb[15:0]*' is the error data output port, '*addra[14:0]*' is the address port for writing the error data to a particular location, '*addrb[14:0]*' is the address port for reading the error data from a particular location, '*clka*' , '*clkb*' are the clock inputs for synchronous operation, '*wea*' is the write

enable port used to control the write operations of the unit ('0' – No Write, '1' – Normal Write operation) and '*enb*' is the enable port for controlling the read operations in the unit ('0' – No Read, '1' – Normal Read operation). The clock inputs '*clka*' and '*clkb*' are connected to the same clock to perform read and write operations at the same clock frequency. The input of the memory unit is connected to the Error Output of the Processor Core and the address locations at which the data needs to be read from or written into is controlled by the Processor Core Control Unit. The output from this unit is fed back to the Error Filter circuit in the Processor Core.

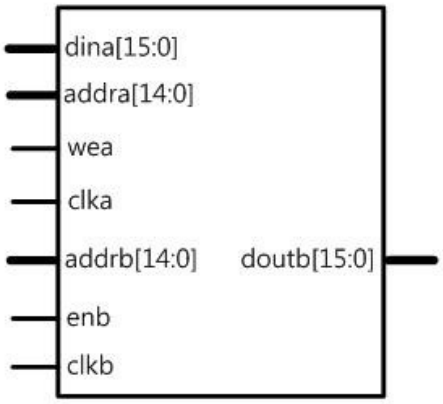

Figure 6.1: Error Storage Block

RAM Memory Schematic

```
for (m=0; m<bufferSize; m++)
{
 for (n=0; n<imageCol; n++)
 {
  for (c=0; c<imageLvl; c++)
  {
   errorImage[c+n*imageLvl+m*imageCol*imageLvl]=((float)rand()/(float)RAND_MAX-0.5)*noiseLevel;
  }
 }
}
```

Figure 6.2: Code Snippet Showing Random Values Stored in the Error Image Buffer

The code snippet in Figure 6.2 shows that initially all the memory locations are filled with some random value, this is shown in the software code in terms of '*errorImage[]*'. This is achieved by initializing all the memory locations to some random values with the help of a *.COE* file shown in Figure 4.10. The value for '*noiseLevel*' in the code is a constant value of 0.1. The initialization values are generated from the same 'C' code and

converted to a binary format that can be loaded into the hardware memory unit. The address ports in this unit are connected to an address counter unit that increments or decrements the address depending on the pixel location. If '*n*' is the number of columns (width) of an image, then '*(n-2)*' error values need to be stored. This is explained in detail with respect to the image shown in Figure 6.3. Generally in any image, errors are produced at every pixel location and the average number of error updates per pixel location is 3. But, for cases discussed in Figure 3.4, the number of error updates comes down to 2. The terms '*A*' and '*B*' in the Figure 6.3 provide important information about how the stored errors are added to the corresponding pixel locations. The stored error at a pixel value should be added with the error-filter weight [7/16] and sent to the Previous Pixel Register for processing. '*A*' indicates the errors being read from the storage unit and '*B*' indicates the errors generated at each pixel location being stored in the memory unit. It can be observed that all the error values from the pixels are stored in the memory storage except the last 2 pixels. The gray boxes indicate the 16 bit register associated with the weight filter [7/16] in the Processor Core which is shown in Figure 5.21 in which the register takes either the input from the Error Storage Register or the value from Adder [3/16].

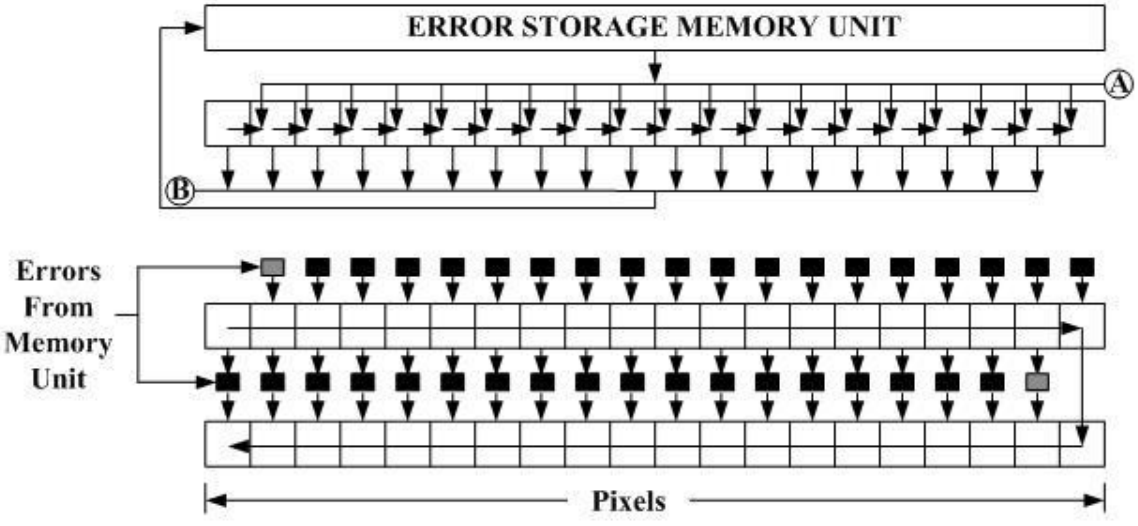

Figure 6.3: Error Storing Procedure Schematic

This process is better shown by the black boxes that represent the errors produced at each pixel. The process of reading and writing the errors occur simultaneously. Referring to the Figure 1.4, the last pixel in a row doesn't have the Error-Filter weights [7/16] and [1/16]; the error at this location needs to be added to the value of the next pixel. Thus, this error is connected directly to the Previous Pixel Register and this is the reason why the error is not stored in the Error Storage Memory Unit. The read and write addresses are connected to the same address counter and the storage unit is ingeniously designed in such a way that the address is the same for reading and writing at any point of time and the only difference is that the data is not read or written to the memory unit at the same clock edge. This prevents collisions that may occur if an address location is read and written at the same time. Since the system implements a serpentine scan technique, the counter must be able to count up and down depending on the image row being processed. The error corresponding to a pixel location must be added with the error-filter weight [7/16] of the previous pixel and the result is to be sent to the Previous Pixel Register of the current pixel.

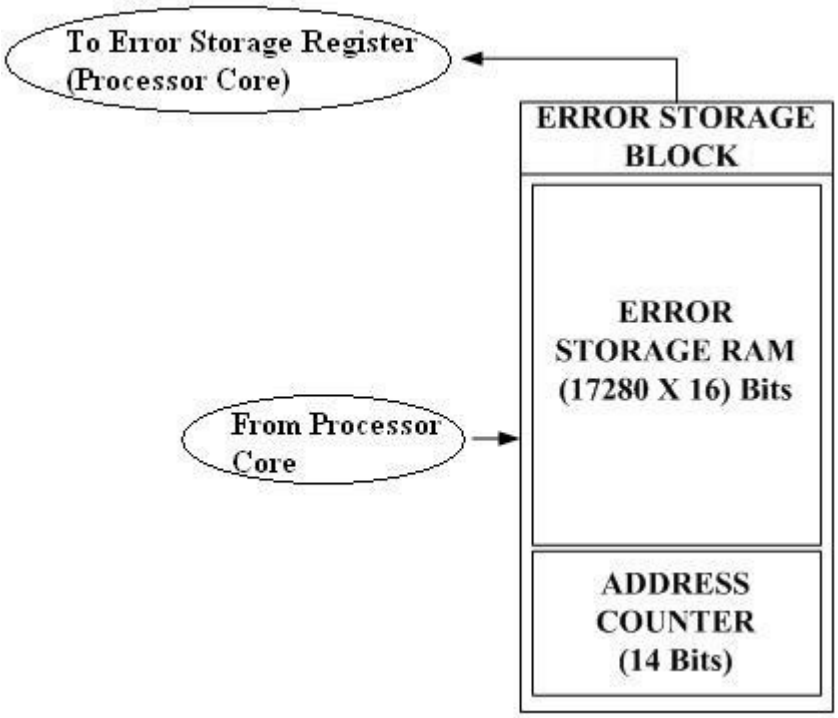

Figure 6.4: Error Storage Block RAM Memory Unit

Figure 6.4 shows the high level schematic of the Error Storage Block RAM Memory Unit which has two data ports, one from the Processor core and the other to the Core. The address generation and control is discussed in detail in the next section which includes the design and implementation of the address counter for the memory unit including the Input Image Size Monitor.

6.3 **Input Image Size Monitor**

This section deals with the design and implementation of the unit used to count the number of rows and columns of an image being processed. The Image Size monitor is a binary counter with some combinatorial circuits added to control other units in the whole system. The input to this counter is the output from Parameter Register 2 which gives the number of Rows and Columns in the input image to be processed. This counter controls the Address Counter of the Error Storage Memory Unit indicating when to count up or down. If the current row being processed is odd, the Image Size Monitor instructs the Address counter to count up and if the row is even then it instructs the counter to count down thus establishing a serpentine scan technique. Figure 6.5 shows the higher level schematic of the Image Size Counter circuit where '*cin[15:0]*' represents the number of columns (image width), '*rin[15:0]*' represents the number of rows (image height), '*up*' is the control bit that instructs the counter to count up ('0' – no count, '1' – count up), '*clr*' is the clear bit to reset the counter initially to a known value, '*clk*' is the clock input, '*FCOL*' is the output port that indicate whether it is the First Column (pixel) in a row, '*LCOL*' indicates whether it is the Last Pixel in the row, '*LROW*' indicates whether it is the Last Row being processed and '*up_err*' is the control bit connected to the Address Counter of the Error Storage Memory Unit ('0' – count-up, '1' – count-down). This counter counts up for odd-numbered rows and counts down for even-numbered rows. The address range for up-count is '0' to '[rows – 2]' and for odd rows is '[rows – 2]' to '0'. The Image Size Counter is incremented by the Processor Core Controller depending on the row to be processed and is a positive edge sensitive digital circuit.

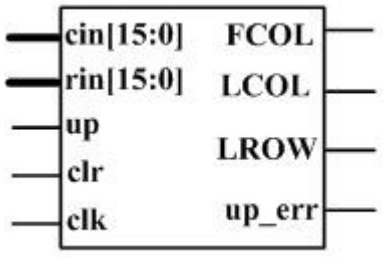

Figure 6.5: Image Size Counter Schematic

6.4 **Error Storage Memory Address Counter**

The Address Counter is the digital component that provides addresses to the Storage memory unit for parallel reading and writing of the errors generated at each pixel of the image. The schematic for the address counter is shown in Figure 6.6 where '*ce*' is the clock-enable input port that controls the operation of the counter ('0' – no operation, '1' – normal operation), '*clr*' is the clear bit, '*enr*' is the read-enable bit ('0' – Read, '1' – No Read), '*enw*' is the write-enable bit ('0' – Write, '1' – No Write), '*up_dn*' is the control bit used to instruct the counter to count up or down depending on the row number ('0' – count-up, '1' – count-down) and also this port is connected to the output port '*up_err*' of the Image Size Counter, '*clk*' is the clock input for synchronous operation, '*addr[14:0]*' is the address port connected to the '*addra[14:0]*', '*addrb[14:0]*' of the Error Storage Memory, '*rd_en*' is the read-enable output port connected the '*enb*' port of the Error Memory and '*wr_en*' is the write-enable output port connected to '*wea*' port of the Error Memory. The ports '*enr*' and '*enw*' are directly connected to '*rd_en*' and '*wr_en*' with a delay circuit in between to transfer the control signal from the controller to the Error Memory at the correct clock edge as shown in the Figure 6.7.

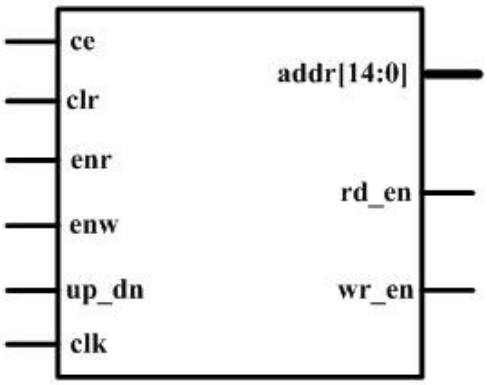

Figure 6.6: Error Storage Block RAM

Memory Address Counter

The controller provides the read/write commands to the Error Memory approximately two clock cycles ahead and in order to transfer the control at the correct clock edge, a delay unit is introduced. The delay unit is a positive edge triggered D-flip-flop and two of these elements are used. The controller is a negative edge triggered system and gives the output after the negative edge, and this is captured at the positive edge by the Address Counter and at the next positive edge by the Error Storage Memory unit. The unit is designed very carefully meeting the set-up and hold time constraints. All the units in the Error Storage Memory Architecture are run at 50 MHz, the same as the overall system frequency. All the units were fully tested and verified with the help of the ModelSim simulation software.

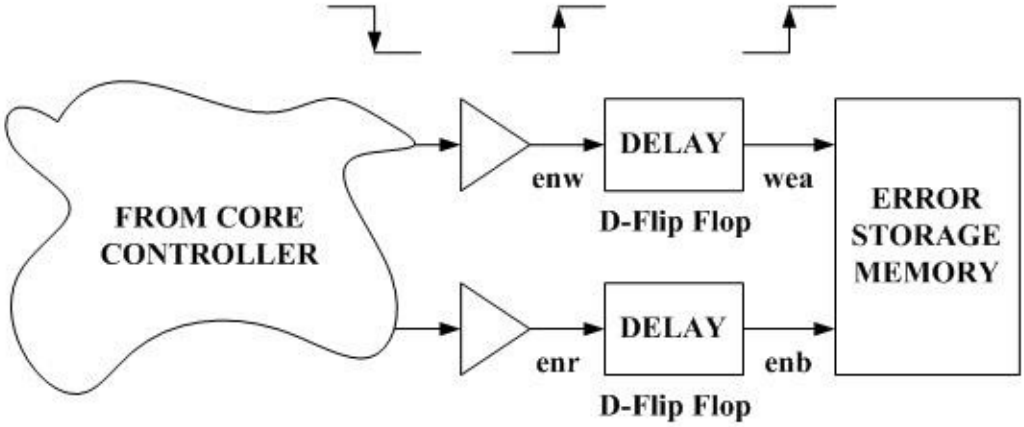

Figure 6.7: Read & Write Port Connections

6.5 Total Functional View of Single Error Storage RAM Memory Module

Figure 6.8 shows the entire architecture of the Error Storage Block RAM memory unit. The Input Image size monitor is connected to the Error Storage Block RAM Address Counter unit that generates the address to which the data is to be stored and read. The error values produced in the Processor Cores are sent to the Error Storage Block RAM and the error values corresponding to a pixel location are accessed from the Block RAM memory. The Error Storage Block RAM is a positive edge sensitive unit and the Error Storage Block RAM Address Counter is negative edge sensitive.

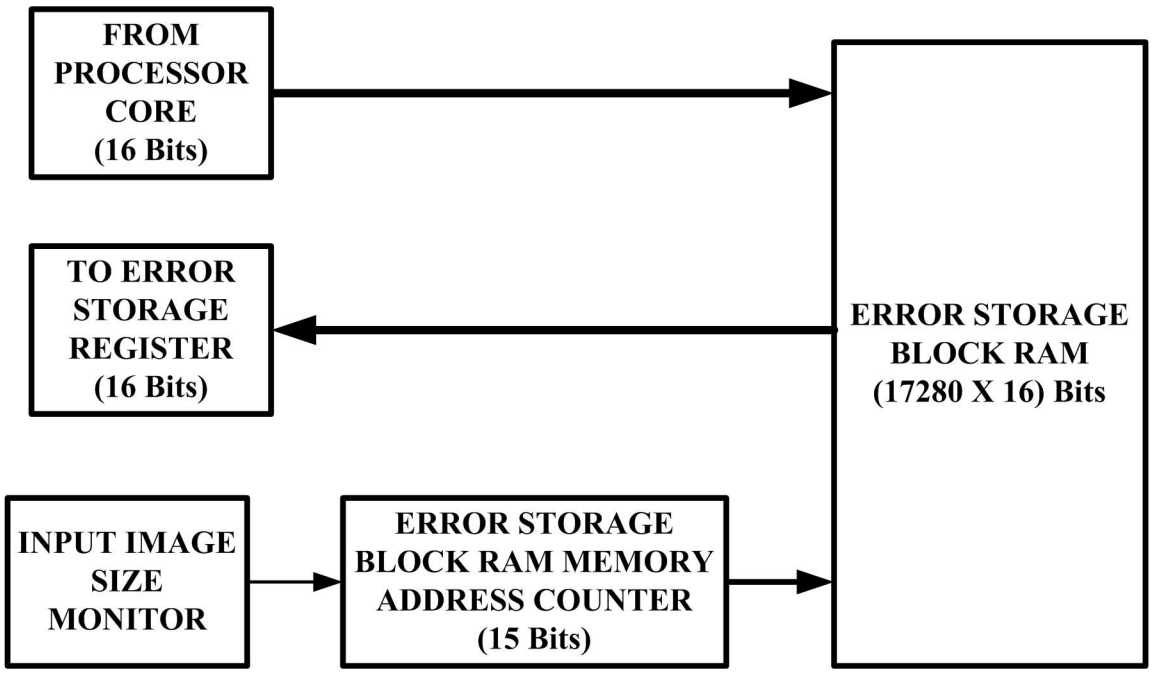

Figure 6.8: Error Storage Block RAM Memory Functional Architecture

Chapter 7. Output System Architecture Design

7.1 Introduction

This chapter addresses the development and design of output functional units of the System Architecture of Figure 3.1. It gives a detailed explanation on the Output Data FIFOs and Output Logic Units used in this Halftoning Architecture. The output unit is one of the most critial units in the architecture as the output pixels generated by the Processor Cores need to be buffered accordingly and the effective output image value is to be calculated with the help of an output logic circuit. All the functional elements in the Output System Architecture are described using Verilog HDL and simulated using ModelSim simulation software.

7.2 Output Data FIFO

Each Output Data FIFO [1-12] of Figure 3.1 is a small memory unit connected to the output of a Processor Core Unit to collect the output bits. This FIFO has 2 address locations that supports data width of 1 bit. The reason of the FIFO having only 2 address locations can be explained with respect to the Figure 3.7 where the gray colored boxes indicate the output produced by each Processor Core. Cores 1 to 4 produce the output of the next pixel when Core 12 delivers the output of the previous pixel. Thus, a FIFO with 2 address locations can accommodate and buffer the output bits without any loss. Figure 7.1 shows the schematic of the Output Data FIFO where '*data_in*' is the input port connected to the Processor Core, '*data_out*' is the output port of the FIFO unit, '*clr*' is the clear bit, '*rd_en*' is the control bit of the FIFO that dominates the read operation ('0' – No Read, '1' - Read), '*wr_en*' is the write control bit of the FIFO ('0' – No Write, '1' – Write) and '*clk*' is the clock input. The output from the FIFO is taken every 8 clock cycles and the control is given by the Core Controller. This is a fully automated process leading to efficient buffering of output pixels. Figure 5.6 shows the software code snippet where the

term '*outputImage[index]*' represents the equivalent hardware memory that is the Output Data FIFO.

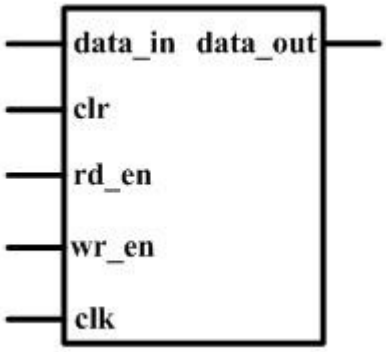

Figure 7.1: Output Data FIFO Schematic

7.3 **Output Logic Unit**

```
j=0;
for (m=0; m<imageCol; m++)
{
 for (n=0; n<imageChn; n++)
 {
  levels=numLevelsPerChannel[n];
  pixel=0;
  for (l=0; l<levels; l++)
  {
   pixel+=(outputBuffer[j]>0.5);          - C-16
   j++;
  }
  inputBuffer[n+m*imageChn]=pixel;
 }
}
```

Figure 7.2: Software Code Snippet for Output Calculation

This unit shown in Figure 3.1 is the most important unit for calculating the combined output from the Processor Cores. Figure 7.2 shows the software code snippet for output calculation using 'C' code. The variables in the code '*imageCol*' represent the number of columns (width) of the image, '*imageChn*' represents the number of colors/channels in the

image, '*levels*' represent the number of levels per channel, '*pixel*' is a temporary variable used to calculate the effective output from the Output FIFOs and '*outputBuffer[j]*' is the output stored in a temporary buffer unit for subsequent output value calculation. It also represents the Output Data FIFOs of Figure 3.1 in digital hardware. The value per level in each pixel will be either '1' or '0' and the code suggests adding all the values in the levels in each channel individually. Figure 7.3 shows the equivalent hardware unit for the output calculation where '*I[2:0]*' is the input port connected to three Output Data FIFOs and '*O[1:0]*' is the 2 bit output calculated by the hardware. This unit is a combinatorial circuit that uses Look-up-tables to produce the output. An adder used in the software code is replaced by the LUTs as it is fast, simple and very efficient. The output is 2 bits wide which can support four possible values (0,1,2,3). The ouput logic counts the number of 1's in all the three levels per channel and the value ranges from 0 to 3.

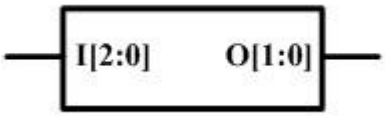

Figure 7.3: Output Logic Unit

Table 7.1 shows the output values according to the input values and these values are stored in memory to access the data according to the input.

Table 7.1: Input Values & Corresponding Outputs

Input (Binary)	Input (Decimal)	Output (Binary)	Output (Decimal)
'000'	0	'00'	0
'001'	1	'01'	1
'010'	2	'01'	1
'011'	3	'10'	2
'100'	4	'01'	1
'101'	5	'10'	2
'110'	6	'10'	2
'111'	7	'11'	3

The Output Logic Circuit is designed using a gate level coding technique for maximum performance and minimum gate delay. Figure 7.4 shows the full Output System Architecture Figure 3.1 (Output Data FIFOs and Output Logic Units) and its connections. The Output Logic Architecture consists of 4 Output Logic Units and 12 Output Data FIFOs (1 per Processor Core). The Cores {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} represent four channels (1 channel per set of Cores mentioned before) and each channel supports 3 levels. The allocation of the processing elements are done according to the channels and starts from the very first set of cores. For example, if there are 3 channels , 3 levels per channel, the sets {1,2,3}, {4,5,6} and {7,8,9} are switched ON for processing. The output obtained is 2 bits per channel per pixel and the total is 8 bits per pixel. As the output logic unit is a combinatorial circuit, the output of the whole system will be obtained in under one clock cycle after the Output Data FIFO provides the output data. For maximum quality in the output image, the number of levels per channel must be 3. The stacking constraint as discussed in Section 1.1.7 is applicable to the elements in each set of channels but not between the sets (no constraint between channels). The throughput in this architecture doesn't depend on the number of channels and levels used, it is the same for any number channels and levels. The output can be directly connected to a printer head for image reproduction using the processed output pixels. The output image pixels can also be stored inside the FPGA using a Block RAM unit for further buffering to other printing devices.

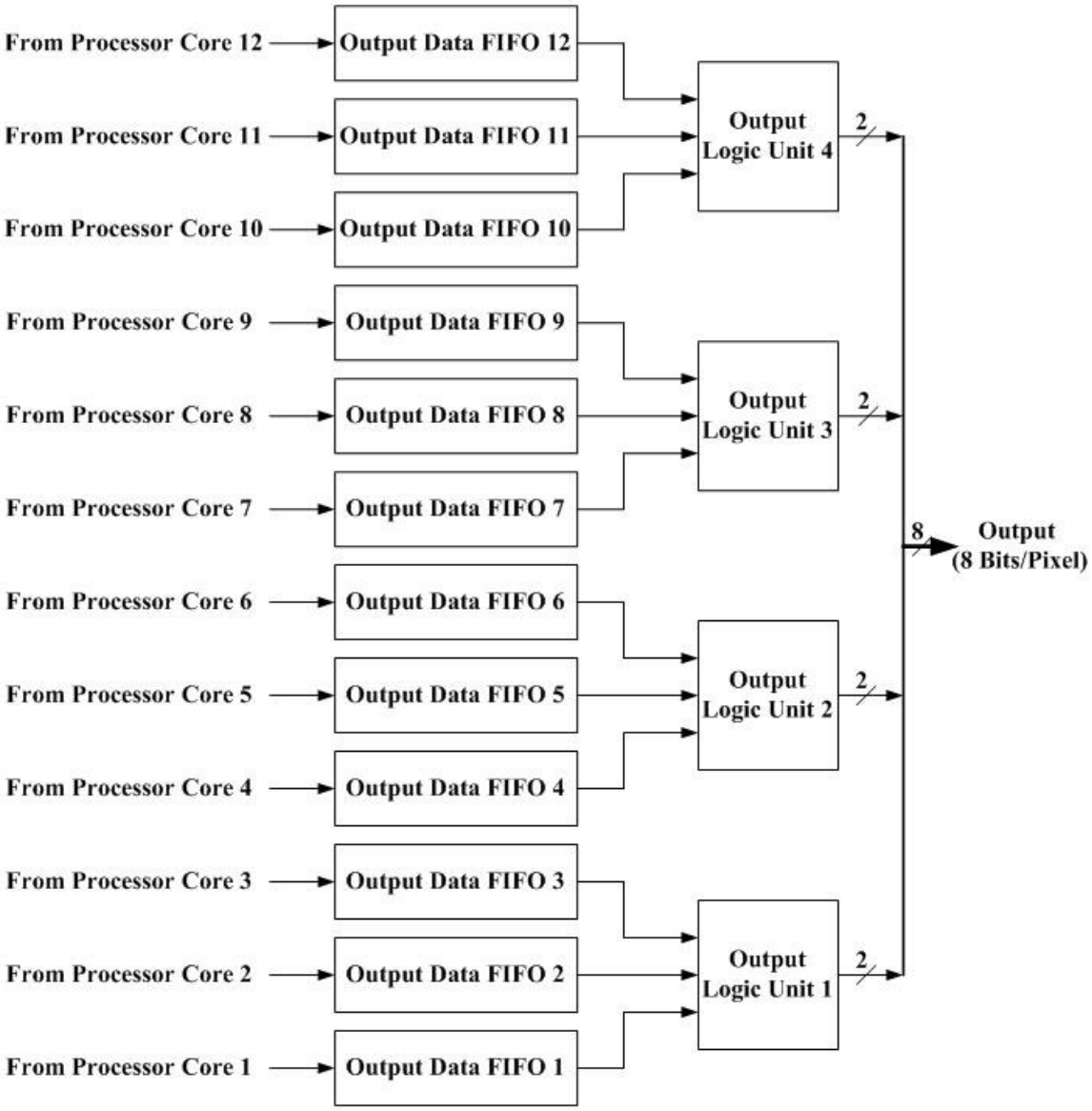

Figure 7.4: Entire Output System Architecture

Chapter 8. Controller Architecture Development and Design

8.1 Introduction

The previous chapters in this research work dealt with the Datapath architecture where the digital elements responsible for the arithmetic, logical and storage operations were discussed. In this chapter a very detailed explanation of the control logic design is presented. In fact, the most challenging and critical part of the system is the controller design. The controller fully automates the system and completely reduces control constraints that arise. The controller can be designed only when the Halftoning algorithm at hand is thoroughly understood and when all constraints are known. The main responsibility of the control logic is to provide command signals for specifying operations to be performed at each system clock cycle. The controller will be a Finite State Machine type which can be defined as a digital logic system that has a fixed number of states and follows a predefined procedure. A 'State' in the Finite State Machine Controller is an entity that defines the operation of a digital element (functional unit) on a particular clock cycle of the system clock. A controller needs to know the previous, present and next state according to the inputs given to the circuit. Thus a Finite State Machine is a combination of both sequential and combinatorial elements. A control logic fully controls all the elements in a digital circuit and leads to production of correct output from the unit. The chapter provides all information about the controller used to entire the whole Halftoning system.

8.2 Mealy and Moore State Machines

Finite state machines fall under two categories, one is the Mealy Model [31] and the other is the Moore Model [31]. In the Mealy model, the output of the State Machine is dependent upon both the input and the present state of the control logic system. Figure 8.1 shows the schematic of both Mealy and Moore models in which the Mealy model output (Control Signal) are a function of both the input and present state whereas the Moore

model output is a function of the present state alone. The Moore state machine is easier to implement and design when compared to the Mealy model as it is dependent on the present state only, so less circuit dependency exists. The Mealy model consumes less states to build since the next state is dependent on the input and present state, there will be less memory required to store the value of previous and next states. The output in a Mealy model is sensitive to the input irrespective of the clock edge. The output can change when the input changes. In the case of the Moore model, the output changes only on the next clock edge. A control algorithm can be modeled by using either a State Transition Table or State Transition Diagram. A State Transition Table describing a Finite State Machine shows the values of input, present state, next state and output values of the Controller. The State Transition Diagram gives a schematic representation of all the states and their transitions from one state to the other including outputs. Thus for every controller logic, there exists a State Table and the State Transition diagram can be drawn using the data obtained from the State table. There are several coding techniques used to design a control unit and they are selected depending on the application. The next section shows the various ways to design a control unit.

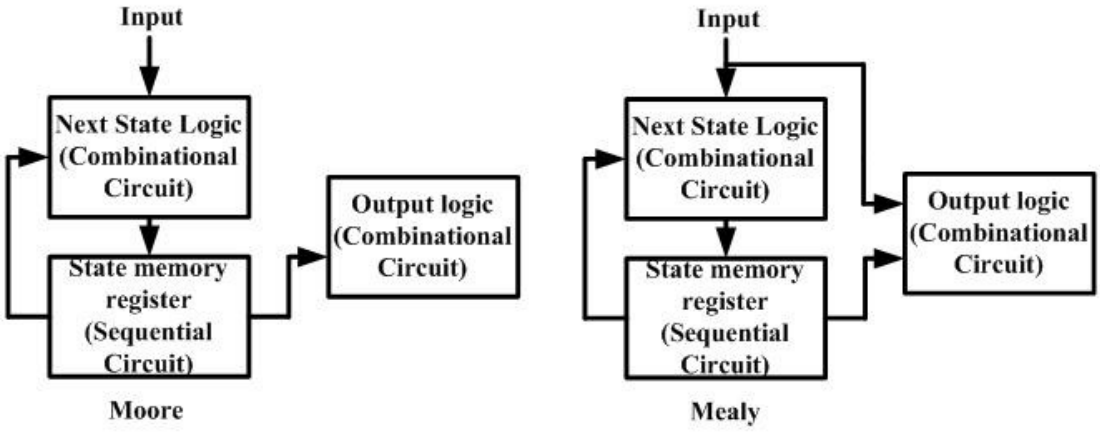

Figure 8.1: Mealy & Moore Models

8.3 **Controller Design Techniques**

The design technique is chosen according to application constraints. Some of the constraints are speed of the system, size of the system and desired efficiency of the unit. The major objective of the control logic design is to build a hardware circuit that achieves

the desired control algorithm in a coherent and uncomplicated procedure. Some types of encoding a Finite State Machine are One-Hot encoding [35], One-Cold encoding [35], Binary encoding [35], Gray Encoding, Almost One-Hot [35], Almost One-Cold [35], Sequence Register and Decoder [34], PLA control [34] and Microprogramed control [34]. The following sections briefly discuss about each of these techniques.

8.3.1 One-Hot Encoding

This method uses one flip-flop per state in the control circuit. The term 'One-Hot' means that only one flip-flop is set to ('1') at any particular time. The control bit is transferred from one flip-flop to another at each clock cycle. The number of flip-flops used is equal to the number of states which results in more flip-flop consumption than any other method. This technique is not useful for Large Scale Integrated circuit implementation. The One-Hot encoding technique is one of the fastest, simplest to build (both combinatorial & sequential), the output logic is very simple to implement and includes only 'OR' gates. This research project will use a One-Hot Encoded type Controller for best performance, noise reduction and simplicity of implementation. Figure 8.2 shows an example of the One-Hot encoding technique for a 7 state Finite State Machine. It can be inferred that each of the 7 states have one flip-flop. There is a separate block for input logic, next state logic and output logic. The output logic contains only 'OR' gates as the output is equivalent to the output from one or more of the D-Flip-flops. Thus this coding technique is simple and efficient.

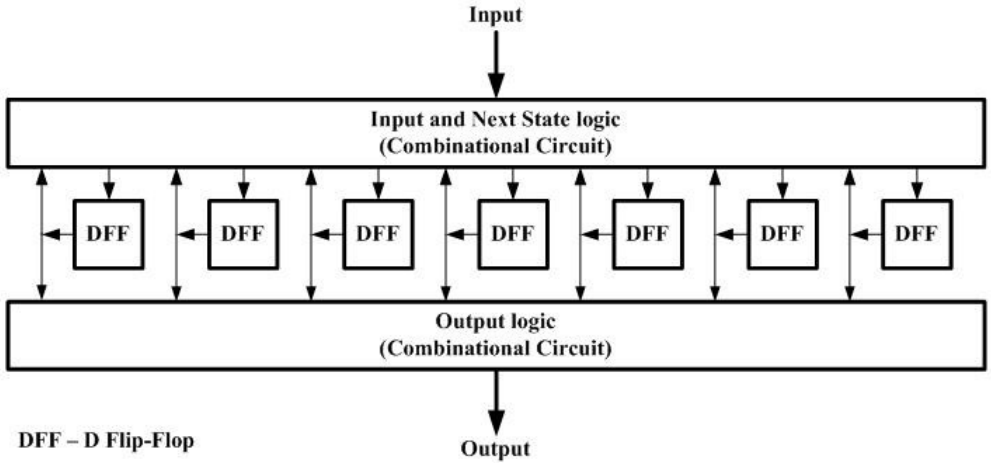

Figure 8.2: One-Hot Encoded Control Logic

8.3.2 Almost One-Hot Encoding

This method is the same as One-Hot except that it takes a bit less than the one-hot. For example, let 'n' be the number of bits/states in a control logic, One-Hot takes 'n' flip-flops to implement the logic whereas Almost One-Hot takes '(n-1)' flip-flops to implement the same logic. This is done by using all zero's to represent a state (typically Initial or Clear State). The performance is the same as compared to the One-Hot but reduces the number of flip-flops to represent a control logic.

8.3.3 One-Cold Encoding

This technique is similar to One-Hot encoding which uses one Flip-Flop to represent a state but the flip-flop currently at work is cleared or set to '0' and all the others are set to '1'. This coding technique has all the attributes of One-Hot encoding and gives the same results.

8.3.4 Almost One-Cold Encoding

This method is similar to Almost One-Hot where it takes a bit less compared to One-Hot. Here the Almost One-Cold also takes a bit less when compared to the One-Cold technique where there exists a state in which all the flip-flops have 1's for a clear or initial state.

8.3.5 Binary Encoding

This type of encoding uses a minimum number of flip-flops depending on the number of states in the given control algorithm. For example, if an algorithm has 7 states, the logic requires 3 flip-flops to accommodate the whole sequence ($2^3 = 8$). Thus binary encoding is proportional to the power of 2. This technique uses the minimum number of flip-flops per range of states. With 'n' flip-flops, 2^n states can be implemented. Figure 8.3 shows the sequence of states in binary encoding for a 7 state control algorithm. The output potentially can contain glitches as there can be more than 1 flip-flop changing state per clock edge and the combinatorial logic circuit is also more complex when compared to One-Hot encoding.

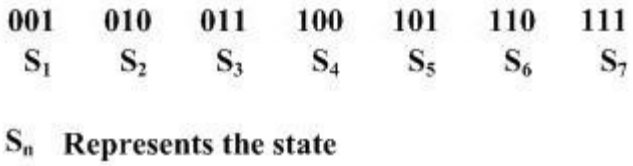

Figure 8.3: Binary Encoded State Machine

8.3.6 **Gray Encoding**

This type of encoding works on the principle of gray code where only one bit out of 'n' changes at a given point of time or clock edge. Gray encoding overcomes the disadvantages of binary encoding where there occurs a lot of glitches and the logic required is reduced. This encoding is useful when the outputs are utilized asynchronously. Figure 8.4 shows the sequence of states in Gray coding where only one bit changes per clock cycle. The number of states that this technique can represent is the same as the binary encoding method except the implementation is different. As only one bit changes, the next state logic and the output logic utilizes less hardware when compared to the Binary state machine.

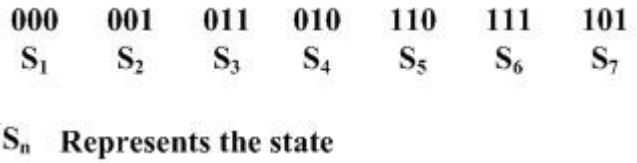

Figure 8.4: Gray Encoded State Machine

8.3.7 **Sequence Register & Decoder Technique**

This method is used in Medium Scale Integrated circuits where the techniques discussed previously are not so efficient and feasible. This method uses a register to transition through the states and the output of the register is connected to a decoder to provide the outputs. If 'n' flip-flops are used in the sequence register, it can support 2^n states and the decoder provides 2^n outputs as well. Figure 8.5 shows the schematic of this technique where the input logic unit decides which state to go to according to the output of the decoder. The output from the decoder is taken as the present state logic and compared with the input logic to get the next state value. Thus all the control signals required to control a system can be generated using this technique.

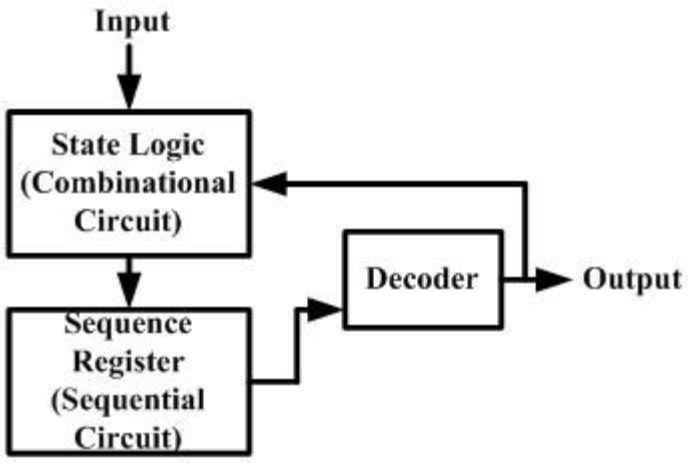

Figure 8.5: Sequence Register & Decoder Technique

8.3.8 **PLA Control**

PLA is the acronym for Programmable Logic Array which is a device used to implement complex digital circuits. It is a Large Scale Integrated (LSI) circuit that can be used to design large complex combinational circuits efficiently. This method is similar to the Sequence Register and Decoder method but all the combinational circuits are implemented using a PLA. The PLA logic reduces the hardware logic and decreases the routing complexity. Figure 8.6 shows a PLA based controller where the Sequence Register provides the present state information and the PLA connected to the input along with the sequence register decides the microoperations to be performed. This control method is used in circuits with a complex hardware and which is difficult to control using conventional state machine techniques. For example, for approximately 100 states, One-Hot encoding uses 100 flip-flops and not so feasible for large complex circuit control. Thus, PLA based control comes to play in these circuits which offers a feasible solution to accommodate all the states.

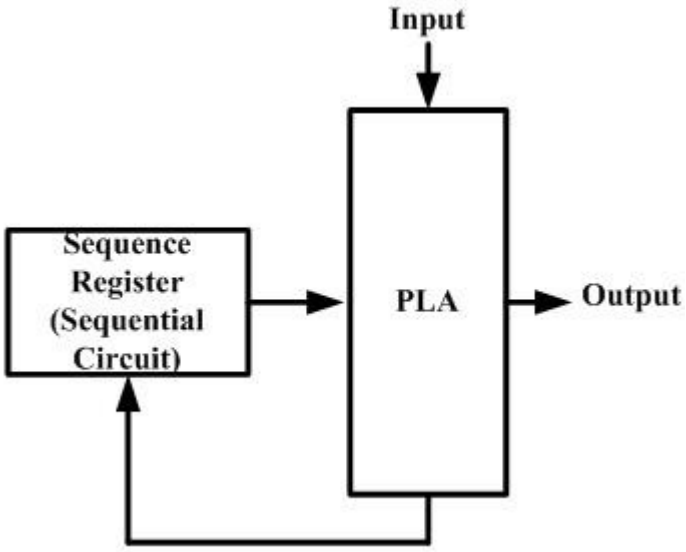

Figure 8.6: PLA Control Technique

8.3.9 **Microprogramed Control**

In this type of control, the control program or sequence is coded into a memory (stored in memory). The memory is typically a ROM (Read-Only Memory) where the control code is hard-coded. This type of control is useful for applications or algorithms in which there is a specific sequence that needs to execute periodically over a long time. Each micro-instruction is stored in an address location and is accessed accordingly at each clock edge. The control algorithm can be updated by simply re-writing the ROM with a new sequence. The control unit consists of an opcode which defines the operation to be performed by the datapath unit. It has a control address register and decoder that selects the micro-instructions. The control address register gives the address location where the specific micro-instruction is located. The micro-instruction field has the address value of the next micro-instruction and the present control sequence. The address of the next micro-instruction is fed to the address register so that the micro-instruction is obtained from the ROM. This type of control technique is used in general purpose processor architectures like Reduced Instruction Set Computers (RISC).

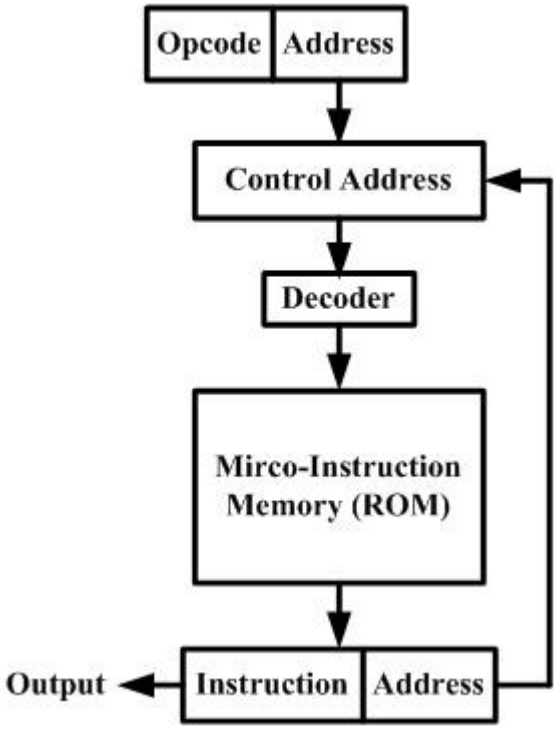

Figure 8.7: Micro-Programmed Control Technique

8.4 System Controller Architecture Strategy

The Halftoning Hardware Architecture has 2 main Controllers for controlling the entire system. It has the Input Memory Controller and the Processor Core Controller. The Input Memory Controller controls the Input Image FIFO, Parameter Registers 1 and 2, Droplet Densities Storage ROM, Input Level FIFOs and the Core Data FIFOs. The Processor Core Controller controls the Processor Cores, Error Storage Block RAMs, Image Size Monitor, Error Storage Block RAM Address Counter, Output Data FIFOs and Output Logic Units. One way to implement a control logic is to replicate the Processor Core Controller depending on the number of Processor Cores. In this research project, only one Processor Core Controller is used and the control outputs are connected to the respective Processor Cores with the help of Control Registers. The data buffering operations are controlled by the Input Memory Controller. Both the Controllers are designed using One-Hot encoding and the following sections discuss their design and operation in detail.

8.5 **Input Memory Controller Design**

The Input Memory Controller unit shown in Figure 3.1 is one of the two crucial controllers that manages the input data transfer to the Processor Cores. This controller controls the Input Data FIFO, Parameter Registers 1 and 2, Droplet Densities Storage ROM, Input Level FIFO and Core Data FIFO as shown in Figure 4.13. The high level schematic of the controller is shown in Figure 8.8 where the inputs are on the left and the outputs to the right. This controller is designed using One-Hot encoding technique. '*C[1:0]*' is the input port that is connected to Parameter Register 2 which gives the number of channels per pixel, '*INIT*' is the port connected to the Processor Core Controller which indicates that all the Cores are ready for processing, '*LVAEMPTY*' is the signal that is connected to the 'Almost Empty' port of the Input Level FIFO, '*LVAFULL*' is the signal that is connected to the 'Almost Full' port of the Input Level FIFO, '*LVFULL*' is the signal that is connected to the 'Full' port of the Input Level FIFO, '*ON_OFF*' is the control switch for this controller, '*RFULL1*', '*RFULL2*', '*RFULL3*', '*RFULL4*' are the signals connected to the 'Full' ports of the Core Data FIFOs (1, 2 and 3), (4, 5 and 6), (7, 8 and 9), (10, 11 and 12), '*START*' is the input port that instructs the Input Controller to start buffering the data, '*clr*' is the reset bit used to clear the controller, '*clk*' is the clock input for synchronous operation, '*op[7:0]*' is the output port that has all the control signals connected to specific ports in the Input Data Memory Architecture and '*STOUT*' is the output that instructs the Processor Core Controller to start processing the data. All the elements this controller manages are positive edge sensitive, thus the control logic is designed to be sensitive at the negative edge of the clock. This prevents any set-up and hold time violations that may occur. The state diagram for the Input Memory Controller shown in Figure 8.9 describes the operations that take place at each state. The following describes the control sequence of the controller. Some variables are used in the state diagram in which '*port*' represents '*port = 1*' and '$\overline{port}$' represents '*port = 0*'.

- 'D_{ON}' is the first state in which the controller remains till it receives the '*ON_OFF*' signal.

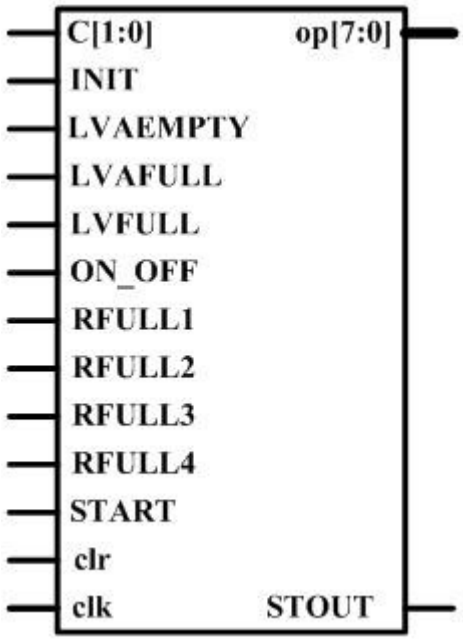

Figure 8.8: Input Memory Controller Schematic

- 'D_{CLR}' is the clear state that resets all the datapath elements controlled by this unit.
- 'D_{RDY}' is the state in which the Input Image FIFO tells the controller to start the data buffering operations. This is done by the port '*READY*'. The controller remains in the same state for '$\overline{READY}$' and moves to the next state for '*READY*'.
- 'D_0' and 'D_1' are the states in the sequence that starts the input memory buffering. The data is read from the Input Image FIFO and sent through the Droplet Densities Storage ROM and finally received by the Input Level FIFOs.
- 'D_2' and 'D_3' are the states that are executed in parallel where the Input Level FIFO is filled with the values from the Storage ROM. The controller keeps reading the Input Image FIFO till the Input Level FIFO is almost full. When '*LVAFULL*' is active, then the controller stops reading from the Input Image FIFO. It is the same in case of the state 'D_3' which is used to write data into Input Level FIFO.

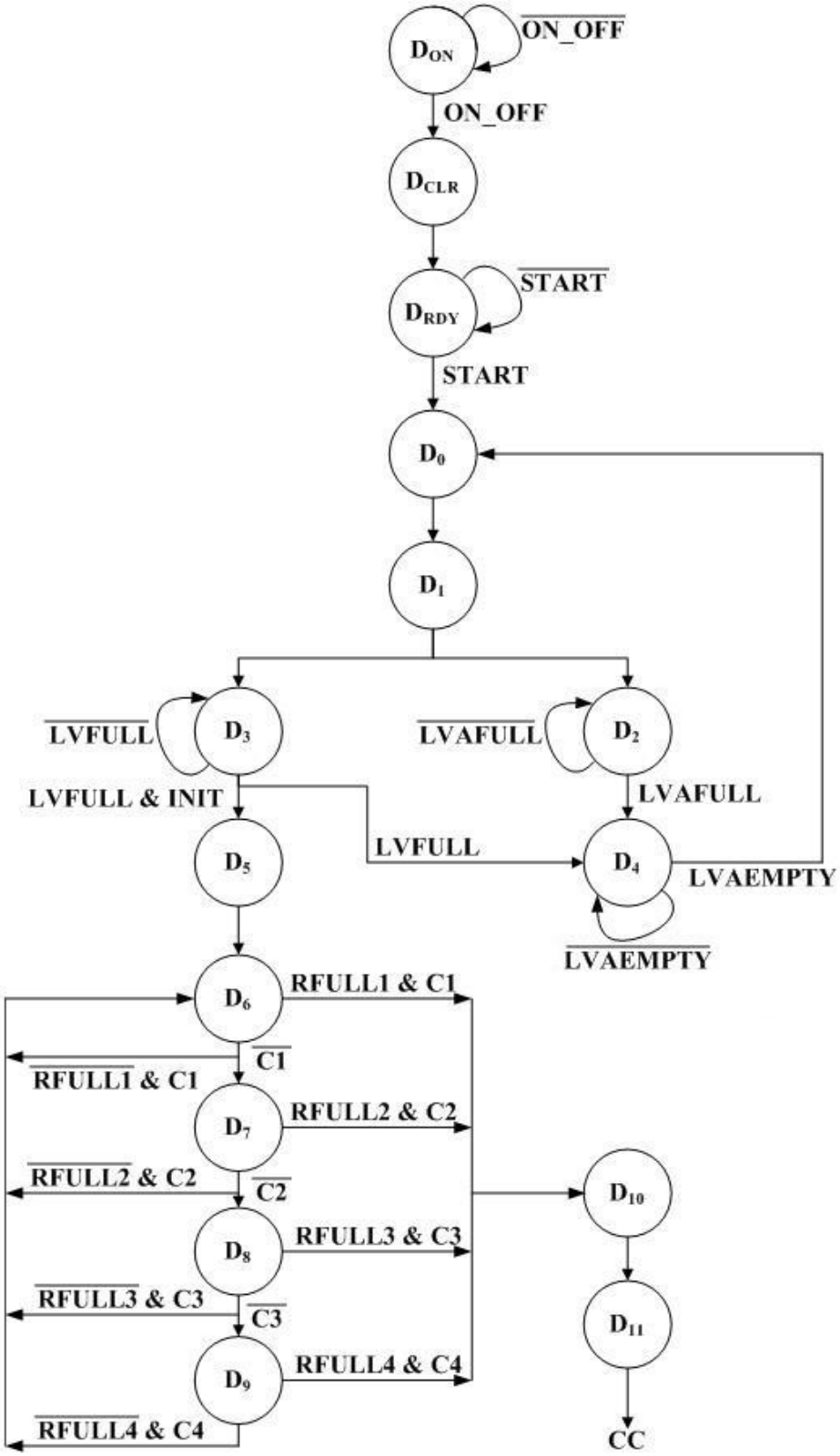

Figure 8.9: State Diagram for Input Memory Controller

- The controller has two small parallel control operations that take place till the final pixel of the image is reached. The controller goes to state 'D_5' when the '*INIT*' or ready signal from the Processor Controller and the Level FIFOs full signal is asserted. In this state, the controller enables the read operation of the Level FIFO as the read or write operations have one clock cycle latency, the read signal must be given to the FIFO one clock edge before enabling the write-enable bit of the Core Data FIFO.

- The state 'D_6', 'D_7', 'D_8', 'D_9' deals with the Core Data FIFO read/write operations. There are 4 states mentioned as the architecture supports up to 4 channels. The term '*C[1:0]*' gives the number of channels in the given image ('*C1*' – 1 channel, '*C2*' – 2 channels, '*C3*' – 3 channels, '*C4*' – 4 channels). The sequence of execution of the four states mentioned depends on the channel count and doesn't depend on the number of levels. For example, if there are 4 colors then, states circulate from 'D_6' –> 'D_7' –> 'D_8' –> 'D_9' till the FIFO is filled. For 3 colors it is 'D_6' –> 'D_7' –> 'D_8', for 2 colors it is 'D_6' –> 'D_7' and for 1 color it is 'D_6'. The Input Level FIFO is read and Core Data FIFO is written simultaneously in all the mentioned states.

- The controller enters state 'D_{10}' when the Core Data FIFOs are almost full. In this state the read-enable signal is deactivated as the reading is one clock ahead of writing. The last location in the Core Data FIFO is filled with the data from the Level FIFO.

- The last state is the 'D_{11}' state which enables the start signal of the Processor Core Controller resulting in pixel processing. The signal connected to the Core controller is represented by '*CC*'.

- The states 'D_5', 'D_6', 'D_7', 'D_8', 'D_9', 'D_{10}', 'D_{11}' are executed only once at the beginning before the pixels are processed. The Core Data FIFOs are automatically filled by a small unit connected to Core Controller. This reduces the complexity and results in the simplest design possible. But the states 'D_0', 'D_1', 'D_2', 'D_3', 'D_4' are executed all the time till the image is completely buffered.

Table 8.1: Control Table showing Outputs and States

Control State	op[0]	op[1]	op[2]	op[3]	op[4]	op[5]	op[6]	op[7]	STOUT
D_{ON}	0	0	0	0	0	0	0	0	0
D_{CLR}	1	0	0	0	0	0	0	0	0
D_{RDY}	0	0	0	0	0	0	0	0	0
D_0	0	1	0	0	0	0	0	0	0
D_1	0	1	0	0	0	0	0	0	0
D_2	0	1	0	0	0	0	0	0	0
D_3	0	0	1	0	0	0	0	0	0
D_4	0	0	0	0	0	0	0	0	0
D_5	0	0	0	1	0	0	0	0	0
D_6	0	0	0	1	1	0	0	0	0
D_7	0	0	0	1	0	1	0	0	0
D_8	0	0	0	1	0	0	1	0	0
D_9	0	0	0	1	0	0	0	1	0
D_{10}	0	0	0	0	1	1	1	1	0
D_{11}	0	0	0	0	0	0	0	0	1

- The outputs of this controller represented by '*op[7:0]*' carries the control signals for the Input Data Memory Architecture. '*op[0]*' is the signal to reset all the datapath elements connected to this controller, '*op[1]*' activates the read-enable bit of the Input Image FIFO, '*op[2]*' is responsible for writing the data from Input Image FIFO to the Input Level FIFO, '*op[3]*' sets the read-enable bit to '1' for reading the data values from the Input Level FIFO, '*op[4]*' writes data from Input Level FIFO to the Core Data FIFO (1, 2 & 3), '*op[5]*' writes data from Input Level FIFO to the Core Data FIFO (4, 5 & 6), '*op[6]*' writes data from Input Level FIFO to the Core Data FIFO (7, 8 & 9), '*op[7]*' writes data from Input Level FIFO to the Core Data FIFO (10, 11 & 12) and '*STOUT*' is the start signal given to the Core Controller for processing the data in the Core Data FIFO. Table 8.1 shows the control table for the Input Memory Controller.

8.6 **Processor Cores Controller Design**

This control unit is the most vital part of the Hardware Architecture that is responsible for flawless processing of the input pixels. The controller is responsible for controlling the Processor Cores, Error Storage Block RAM Memory System and the Output logic System. Figure 8.10 shows the high level schematic of the Core Controller which is positive edge sensitive. This Controller is designed using One-Hot Encoding technique. '*FCOL*', '*LCOL*', '*LROW*' are the input ports of the controller connected to the Input Image Size Monitor that determines the specific pixel location, '*ON_OFF*' is the start signal given to the control unit to initialize, '*START*' is the signal connected to the Input Memory Controller that enables the Processor cores to start processing the input pixels, '*clr*' is used to reset the controller at the start, '*clk*' is the clock input for synchronous operation, '*eop[2:0]*' are the control signals connected to the Error Storage RAM, '*op[23:0]*' are the control signals connected to the Processor Cores, '*cop*' is used to control the Input Image Size Monitor, '*oop*' is the signal to control the output system and '*INIT*' is the signal to the Input Memory Controller stating that the Processor Core Controller is ready for processing. The Figure 8.11 shows the state transition diagram for the Core Controller unit. The operations in various states are described below as follows.

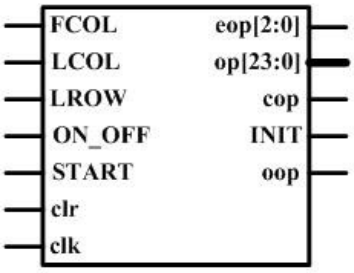

Figure 8.10: Processor Core Controller Schematic

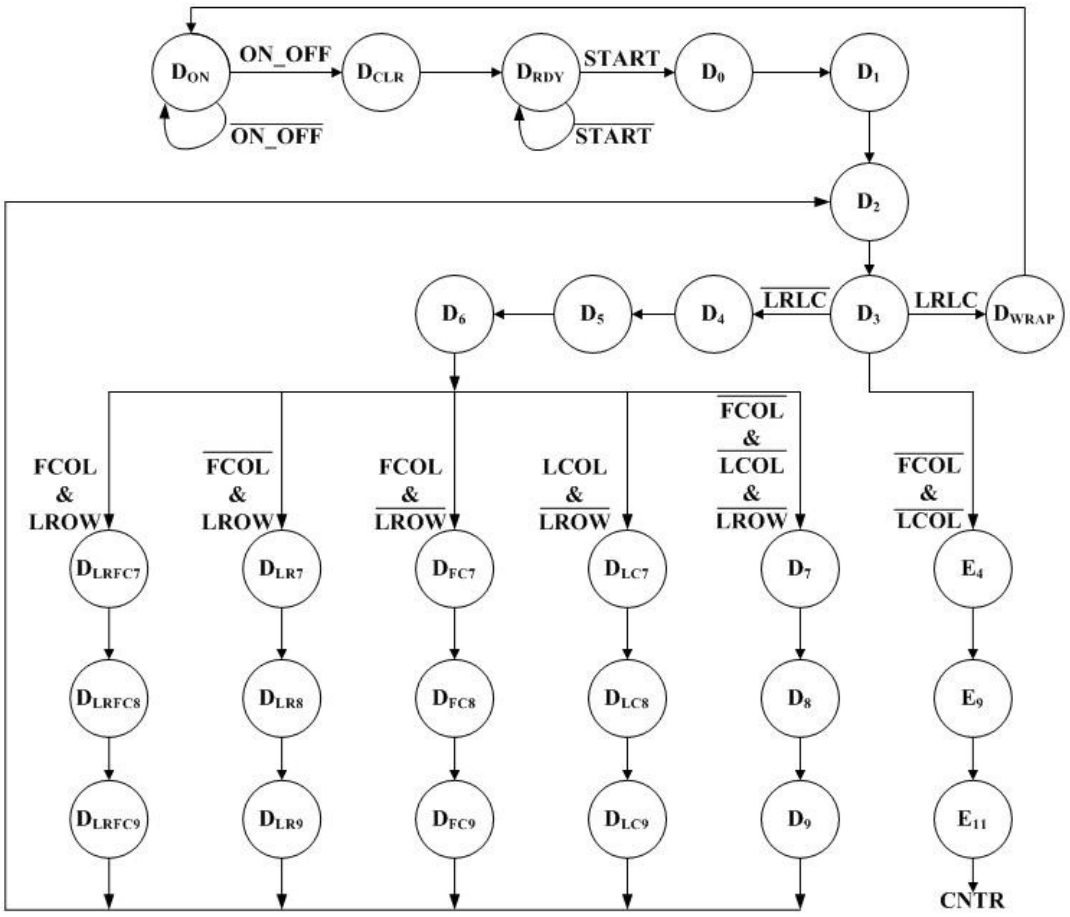

Figure 8.11: Processor Core Controller State Transition Diagram

- 'D_{ON}' is the first state in which the controller remains till it receives the '*ON_OFF*' signal. Both Input Memory Controller and the Processor Core controller are started at the same time (switched -ON).
- 'D_0' is the state where the controller activates the read-enable bit of the Data Core FIFO. The data is read from the Core FIFO.
- 'D_{RDY}' is the state in which the controller indicates that it is ready to accept data. The control unit stays in this state until the '*START*' signal is activated by the Input Memory Controller.
- 'D_1' is the state where the controller instructs the Input Pixel Register and the Previous Pixel register to store the data obtained. The data read from the Core

FIFO is loaded into the Input Pixel Register and the Previous Pixel Register is loaded with a previous data value from the processor core.

- 'D_2' is the control state which notifies the adder to add the values in the two registers (Input Pixel & Previous Pixel).
- In the state 'D_3', the controller activates the threshold comparison circuit which compares the adder output with a constant threshold value. The signal '*LRLC*' informs the controller that it is the last pixel in the last row being processed (final pixel of the image). This state is branched into three other states to perform parallel operations.
- 'D_{CLR}' is the clear state that resets all the datapath elements controlled by this unit.
- 'D_{WRAP}' is the state in which the controller drives the datapath elements to a halt as it will be the final pixel of the image being processed. The controller goes back to the state 'D_{ON}' after 'D_{WRAP}'.
- State 'E_4' is reached after 'D_3' where the controller enables the read operation of the Error storage RAM. The controller enters this loop only when the current pixel being processed is neither the first or the last in a given row.
- 'E_9' is the state where the controller performs the write operation on the Error Storage RAM. The errors generated by the Error-Diffusion unit are stored at this stage with the help of the Core Controller.
- 'E_{11}' is the state that increments the Error Storage Memory Address Counter to read or write the errors in the Error Storage Block RAM. The '*CNTR*' signal is connected to the Memory Address Counter that is responsible for incrementing or decrementing the address depending on the row being processed.
- As long as the final pixel of the image is not reached, the following states are executed. 'D_4' is the state where the Adder-Subtractor unit is activated to subtract the ceiled output value from the original adder value. The result obtained is the error of the particular channel per pixel.

- In 'D_5', the error value is fed to the Error-Limiting-Circuit to prevent the uncontrollable build up of error.
- 'D_6' is the state where the controller instructs the Error Register to store the final error value. The previous error from the Error Storage RAM is also loaded into the Error Storage Register simultaneously. The random weights generator is also activated to produce the weights.
- The controller reaches the states 'D_{LRFC7}', 'D_{LRFC8}' and 'D_{LRFC9}' only when the pixel being processed is in first column and it is the last row of the image.
- The controller reaches the states 'D_{LR7}', 'D_{LR8}' and 'D_{LR9}' only when the core is processing the last row of the image except the first pixel of the last row.
- The controller reaches the states 'D_{FC7}', 'D_{FC8}' and 'D_{FC9}' only when the first pixel of each row is being processed except the last row of the image.
- The controller reaches the states 'D_{LC7}', 'D_{LC8}' and 'D_{LC9}' only when the last pixel of each row is being processed except the last row of the image.
- The controller reaches the states 'D_7', 'D_8' and 'D_9' only when it is neither the first column, last column of a row and last row of the image being processed.
- The states 'D_{LRFC7}', 'D_{LR7}', 'D_{FC7}', 'D_{LC7}' and 'D_7' instructs the multipliers in the Error-Diffusion units to multiply the stored errors with the random weights.
- The states 'D_{LRFC8}', 'D_{LR8}', 'D_{FC8}', 'D_{LC8}' and 'D_8' in the controller performs the addition operation with the previously diffused errors. In these states, the Core Data FIFO is read for processing the next pixel in line.
- 'D_{LRFC9}', 'D_{LR9}', 'D_{FC9}', 'D_{LC9}' and 'D_9' are states where the controller notifies the registers in the Error-Diffusion unit to store the partially processed errors for further processing. The controller directs the Input Pixel Register and the Previous Pixel Register to load the data values. Table 8.2 shows the control table for Processor Core Controller.

Table 8.2: Control Table for Processor Core Controller

CN.ST	op[0]	op[1]	op[2]	op[3]	op[4]	op[5]	op[6]	op[7]	op[8]	op[9]
D_{ON}	0	0	0	0	0	0	0	0	0	0
D_{CLR}	0	1	1	1	1	0	0	0	1	1
D_{RDY}	0	0	0	0	0	0	0	0	0	0
D_0	1	0	0	0	0	0	0	0	0	0
D_1	1	0	0	0	0	0	0	0	0	0
D_2	1	0	0	0	0	0	0	0	0	0
D_3	1	0	0	0	0	0	0	0	0	0
D_4	1	0	0	0	0	0	0	0	0	0
D_5	1	0	0	0	0	0	0	0	0	0
D_6	1	0	0	0	0	0	0	0	0	0
D_7	1	0	0	0	0	1	0	1	1	1
D_8	1	1	1	1	1	0	0	0	0	0
D_9	1	0	0	0	0	0	1	0	0	0
DE_4	0	0	0	0	0	0	0	0	0	0
DE_9	0	0	0	0	0	0	0	0	0	0
DE_{11}	0	0	0	0	0	0	0	0	0	0
D_{LRFC7}	1	0	0	0	0	0	0	0	1	0
D_{LRFC8}	1	1	0	0	0	0	0	0	0	0
D_{LRFC9}	1	0	0	0	0	0	0	0	0	0
D_{LR7}	1	0	0	0	0	1	0	0	1	0
D_{LR8}	1	1	0	0	0	0	0	0	0	0
D_{LR9}	1	0	0	0	0	0	0	0	0	0
D_{FC7}	1	0	0	0	0	0	1	1	1	0
D_{FC8}	1	1	0	1	1	0	0	0	0	0
D_{FC9}	1	0	0	0	0	0	1	0	0	0
D_{LC7}	0	0	0	0	0	0	0	0	0	1
D_{LC8}	0	0	1	1	0	0	0	0	0	0
D_{LC9}	0	0	0	0	0	1	0	0	0	0
D_{WRAP}	0	0	0	0	0	0	0	0	0	0

CN.ST – Control State

Table 8.2 (Continued)

CN.ST	op[10]	op[11]	op[12]	op[13]	op[14]	op[15]	op[16]	op[17]	op[18]	op[19]
D_{ON}	0	0	0	0	0	0	0	0	0	0
D_{CLR}	1	1	0	0	0	1	0	0	0	0
D_{RDY}	0	0	0	0	0	0	0	0	0	0
D_0	0	0	0	0	0	0	0	0	0	1
D_1	0	0	0	0	0	0	0	0	1	0
D_2	0	0	0	0	0	1	1	1	0	0
D_3	0	0	0	0	0	0	0	1	0	0
D_4	0	0	0	1	0	1	0	1	0	0
D_5	0	0	0	0	1	0	0	1	0	0
D_6	0	0	1	0	1	0	0	0	0	0
D_7	1	1	0	0	0	0	0	0	0	0
D_8	0	0	0	0	0	0	0	0	0	1
D_9	0	0	0	0	0	0	0	0	1	0
DE_4	0	0	0	0	0	0	0	0	0	0
DE_9	0	0	0	0	0	0	0	0	0	0
DE_{11}	0	0	0	0	0	0	0	0	0	0
D_{LRFC7}	0	0	0	0	0	0	0	0	0	0
D_{LRFC8}	0	0	0	0	0	0	0	0	0	1
D_{LRFC9}	0	0	0	0	0	0	0	0	1	0
D_{LR7}	0	0	0	0	0	0	0	0	0	0
D_{LR8}	0	0	0	0	0	0	0	0	0	1
D_{LR9}	0	0	0	0	0	0	0	0	1	0
D_{FC7}	1	1	0	0	0	0	0	0	0	0
D_{FC8}	0	0	0	0	0	0	0	0	0	1
D_{FC9}	0	0	0	0	0	0	0	0	1	0
D_{LC7}	1	0	0	0	0	0	0	0	0	0
D_{LC8}	0	0	0	0	0	0	0	0	0	1
D_{LC9}	0	0	0	0	0	0	0	0	1	0
D_{WRAP}	0	0	0	0	0	0	0	1	0	0

CN.ST – Control State

Table 8.2 (Continued)

CN.ST	op[20]	op[21]	op[22]	op[23]	cop	eop[0]	eop[1]	eop[2]	oop	INIT
D_{ON}	0	0	0	0	0	0	0	0	0	0
D_{CLR}	1	0	0	0	0	0	0	0	0	0
D_{RDY}	0	0	0	0	0	0	0	0	0	1
D_0	0	0	0	0	0	0	0	0	0	0
D_1	0	0	0	0	0	0	0	0	0	0
D_2	0	0	0	0	0	0	0	0	0	0
D_3	0	0	0	1	0	0	0	0	0	0
D_4	0	0	0	1	0	0	0	0	1	0
D_5	0	0	0	1	0	0	0	0	0	0
D_6	0	1	0	0	0	0	0	0	0	0
D_7	0	1	0	0	1	0	0	0	0	0
D_8	0	1	0	0	0	0	0	0	0	0
D_9	0	1	0	0	0	0	0	0	0	0
DE_4	0	0	0	0	0	1	0	0	0	0
DE_9	0	0	0	0	0	0	1	0	0	0
DE_{11}	0	0	0	0	0	0	0	1	0	0
D_{LRFC7}	0	0	0	0	1	0	0	0	0	0
D_{LRFC8}	0	0	0	0	0	0	0	0	0	0
D_{LRFC9}	0	0	0	0	0	0	0	0	0	0
D_{LR7}	0	1	0	0	1	0	0	0	0	0
D_{LR8}	0	1	0	0	0	0	0	0	0	0
D_{LR9}	0	1	0	0	0	0	0	0	0	0
D_{FC7}	0	0	1	0	1	0	0	0	0	0
D_{FC8}	0	0	0	0	0	0	0	0	0	0
D_{FC9}	0	0	0	0	0	0	0	0	0	0
D_{LC7}	0	0	0	0	1	0	0	0	0	0
D_{LC8}	0	0	0	0	0	0	0	0	0	0
D_{LC9}	0	0	0	0	0	0	0	0	0	0
D_{WRAP}	0	0	0	0	0	0	0	0	1	0

CN.ST – Control State

8.7 **Processor Core Control Registers**

The Halftoning architecture consists of 12 Processor Cores and each core is one clock cycle behind the succeeding core. The Core Control unit is designed for one core and to control all the cores present in the architecture, there must be 12 control units. This results in higher hardware utilization and is not so efficient. In this research, a high speed and a very efficient control unit is designed which avoids the need for redundant Core Controllers. The main control unit is the Core Controller that is positive edge sensitive and as the Processor Cores are also positive edge sensitive, the control unit for these datapath elements must be negative edge sensitive to eliminate timing problems. Thus, the Core Controller unit is connected to 12 control registers in a sequence as shown in Figure 8.12.

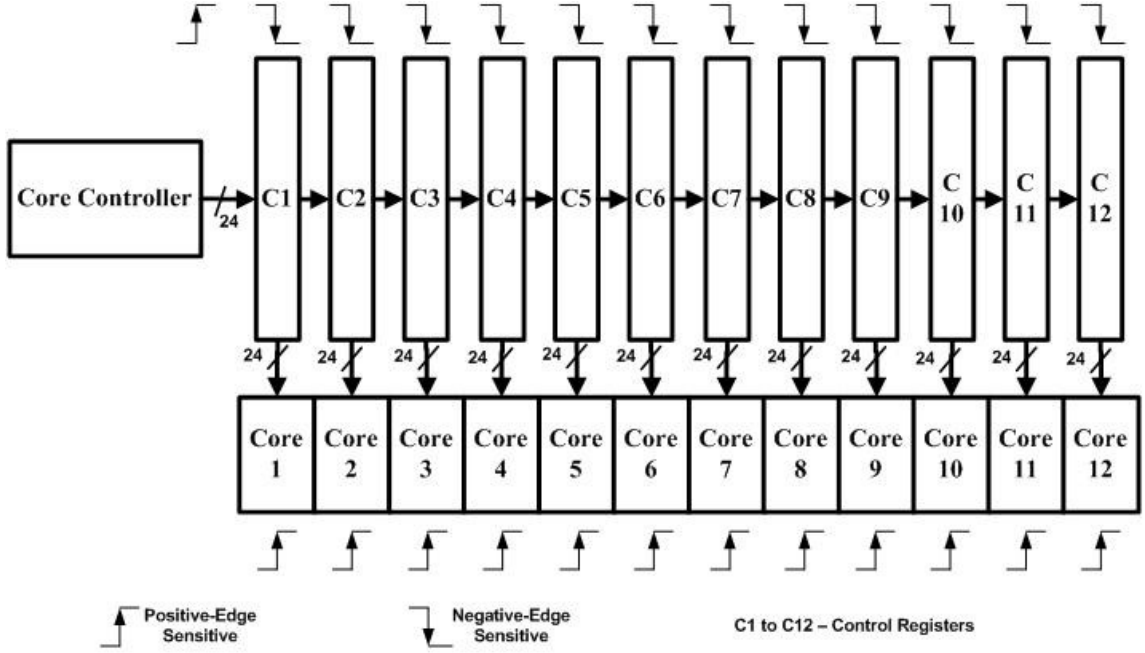

Figure 8.12: Processor Core Control Registers

This process creates a delay between the processing elements and results in the correct execution of the input pixels. The control registers represent a huge shift register shifting its value to the next control register every clock cycle. The control register '*C12*' is 12 clock cycles behind the register '*C1*' and this establishes the one clock cycle delay

between the Processor Cores. There are two different control register designs in this architecture depending on the number of levels per channel used.

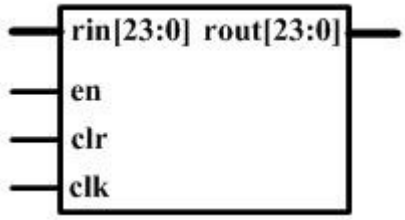

Figure 8.13: Control Register (1 Data Input)

One of the registers is shown in Figure 8.13 where '*rin[23:0]*' is the control input connected to the output of the Core Control Unit, '*rout[23:0]*' is the output of the register which is connected to the control bits of the Processor Cores, '*en*' is the input bit that enables or disables the control register according to the Image parameters, '*clr*' is the reset bit used to clear the contents of the register initially and '*clk*' is the clock input for performing synchronous operations. All the Control Registers are negative edge sensitive.

Figure 8.14: Control Register (3 Data Inputs)

Figure 8.14 shows the other type of Control Register designed to handle the constraints where '*din1[23:0]*', '*din2[23:0]*', '*din3[23:0]*' are the data inputs in which the register can accept the incoming data from 3 different Control Registers, '*addr[1:0]*' indicates which Control Register to accept data from and the rest of the ports are similar to the Control Register in Figure 8.13. This design is used for the cores that support the first level in

each channel with the exception of the first channel. The control bits for the control registers are supplied by a separate unit shown in Figure 8.21.

Figure 8.15 shows how all the control registers are connected. These connections are based on the various image configurations that the Halftoning hardware system supports. The maximum number of channels channels supported are 4 and the number of levels are 3. All the Processor Cores are divided into 4 units with 3 Processor Cores each. The Core '*C4*' uses the 3 data input control register configuration which has inputs from '*C1*', '*C2*' and '*C3*', '*C7*' is connected to the inputs from '*C4*', '*C5*' ,'*C6*' and finally '*C10*' is connected to the inputs from '*C7*', '*C8*' and '*C9*'. These configurations are explained in the points described below where '*I[c,l]*' represents the image with '*c*' channels and '*l*' levels.

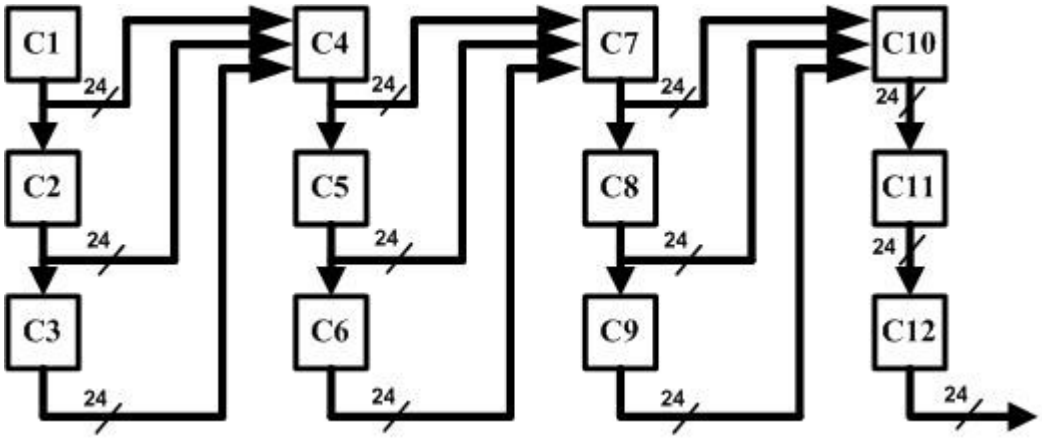

Figure 8.15: Control Register Connections

- For *I[4,3]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C3*' → '*C4*' → '*C5*' → '*C6*' → '*C7*' → '*C8*' → '*C9*' → '*C10*' → '*C11*' → '*C12*' with one clock cycle delay between each Control Register.
- For *I[4,2]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C4*' → '*C5*' → '*C7*' → '*C8*' → '*C10*' → '*C11*'.
- For *I[4,1]*, the control data is transferred from '*CC*' → '*C1*' → '*C4*' → '*C7*' → '*C10*'.
- For *I[3,3]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C3*' → '*C4*' → '*C5*' → '*C6*' → '*C7*' → '*C8*' → '*C9*'.

- For *I[3,2]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C4*' → '*C5*' → '*C7*' → '*C8*'.
- For *I[3,1]*, the control data is transferred from '*CC*' → '*C1*' → '*C4*' → '*C7*'.
- For *I[2,3]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C3*' → '*C4*' → '*C5*' → '*C6*'.
- For *I[2,2]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C4*' → '*C5*'.
- For *I[2,1]*, the control data is transferred from '*CC*' → '*C1*' → '*C4*'.
- For *I[1,3]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*' → '*C3*'.
- For *I[1,2]*, the control data is transferred from '*CC*' → '*C1*' → '*C2*'.
- For *I[1,1]*, the control data is transferred from '*CC*' → '*C1*'.

8.8 **Error Storage Block RAM Control Registers**

The Processor Core Controller controls the operations of Error Storage Block RAMs. Each Processor Core has one Error Storage Block RAM to store the errors generated at each pixel location. There are 12 Error Storage Block RAMs in this hardware design and all of them are controlled by using Error Storage Block RAM control registers. The operating procedure is similar to the Processor Core Control Registers where each register is 1 clock cycle behind its succeeding register. Figure 8.16 shows the high level schematic of the Error Storage Block RAM control registers where '*E1*' though '*E12*' represent the control registers connected in sequence serially. Each register is 3 bits wide and are negative edge triggered to eliminate timing problems.

Figure 8.17 shows the connection diagram of all the Error Storage Block RAM Control Registers. All the Error Storage Block RAMs are divided according to the Processor Cores. The configurations are explained in the points described below where '*I[c,l]*' represents the image with '*c*' channels and '*l*' levels.

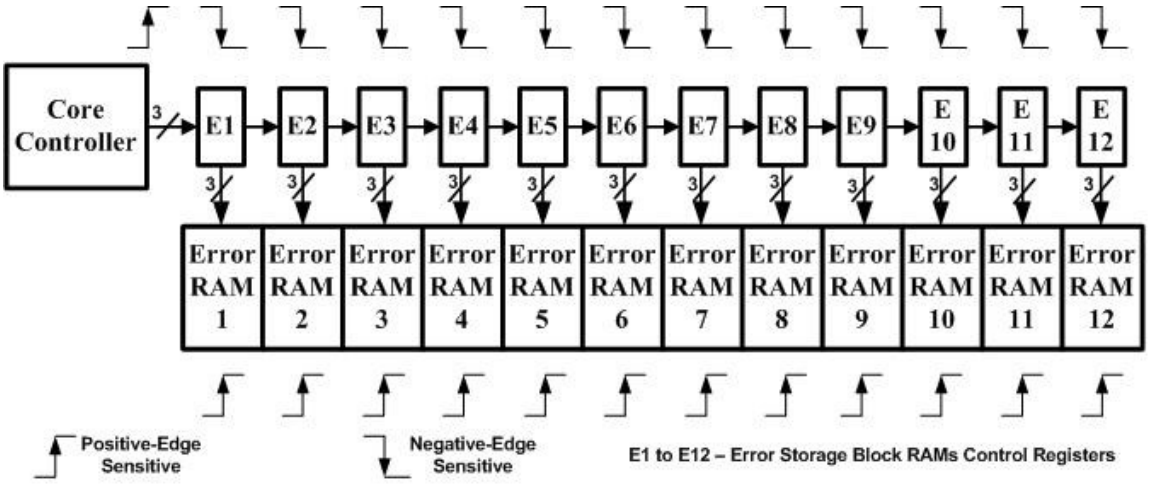

Figure 8.16: Error Storage Block RAM Control Registers

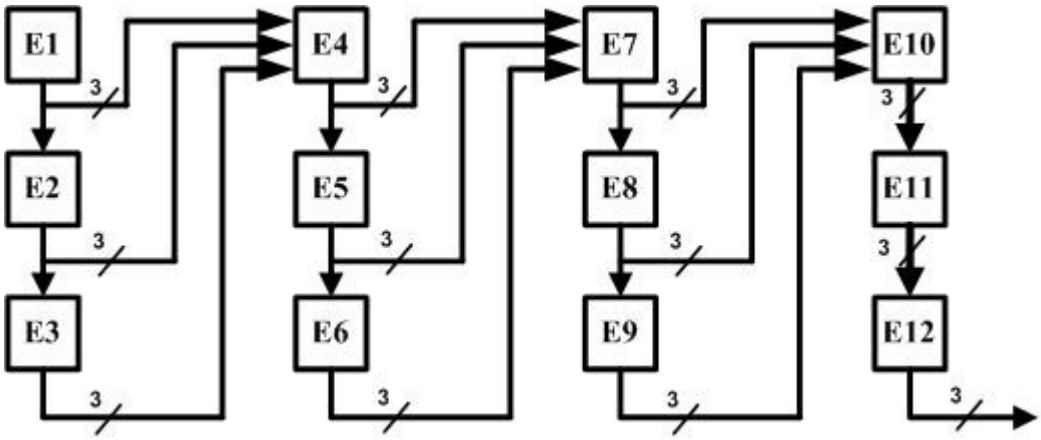

Figure 8.17: Error Storage Block RAMs Control Registers Connections

- For *I[4,3]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E3*' → '*E4*' → '*E5*' → '*E6*' → '*E7*' → '*E8*' → '*E9*' → '*E10*' → '*E11*' → '*E12*' with one clock cycle delay between each Control Register.
- For *I[4,2]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E4*' → '*E5*' → '*E7*' → '*E8*' → '*E10*' → '*E11*'.
- For *I[4,1]*, the control data is transferred from '*CC*' → '*E1*' → '*E4*' → '*E7*' → '*E10*'.

- For *I[3,3]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E3*' → '*E4*' → '*E5*' → '*E6*' → '*E7*' → '*E8*' → '*E9*'.
- For *I[3,2]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E4*' → '*E5*' → '*E7*' → '*E8*'.
- For *I[3,1]*, the control data is transferred from '*CC*' → '*E1*' → '*E4*' → '*E7*'.
- For *I[2,3]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E3*' → '*E4*' → '*E5*' → '*E6*'.
- For *I[2,2]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E4*' → '*E5*'.
- For *I[2,1]*, the control data is transferred from '*CC*' → '*E1*' → '*E4*'.
- For *I[1,3]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*' → '*E3*'.
- For *I[1,2]*, the control data is transferred from '*CC*' → '*E1*' → '*E2*'.
- For *I[1,1]*, the control data is transferred from '*CC*' → '*E1*'.

8.9 **Output Control Registers**

The Output Control Registers manages the operations of the Output System/Logic. There are 12 registers, one for each Output Data FIFO. The Core Controller provides the control signals to these registers and the operation procedure is same as the Core Control Registers. Figure 8.18 shows the ideal connection of all the Output control Registers where the controllers are connected in sequence one after the other. The Output Register Control Circuit consists of 3 main components namely 1-Input Output Control Register, 3-Inputs Control Register and an Output Switch Circuit. The Core Controller gives a single bit control signal to the output registers and depending on the configuration of the input image, the control data is transferred from one output register to the other.

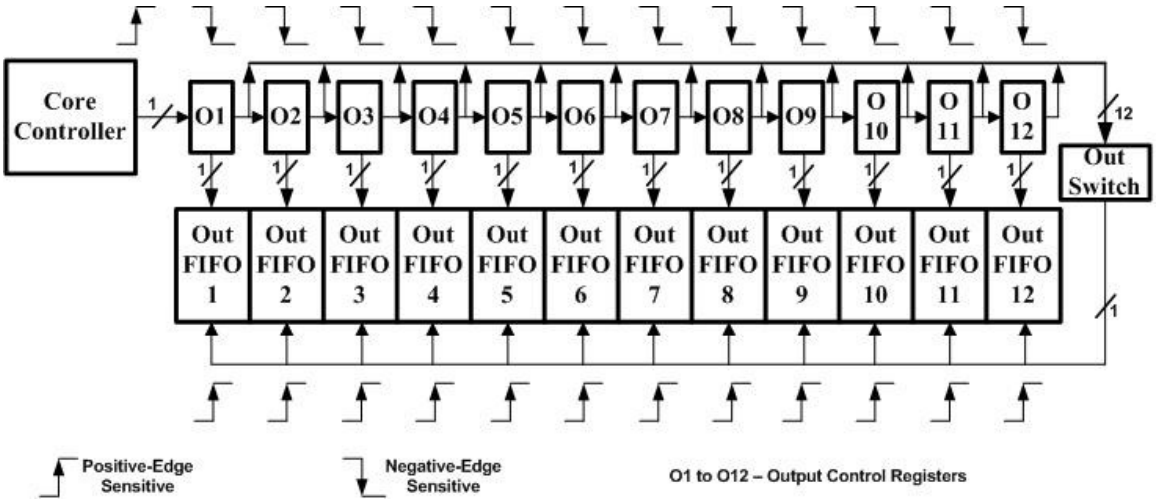

Figure 8.18: Output Control Registers

Figure 8.19 shows the schematic of the three components that constitute the Output Control system where '*inpt*', '*inpt[3:1]*', '*inpt[12:1]*' is the control input ports of 1/3 bit Output control Registers and Output Switch, '*en*' is the port used to enable or disable the Control Register according to the Image Configuration, '*addr[1:0]*', '*addr[3:0]*' indicates the control register from which the data is to be obtained (3 bit Output control Register & Output Switch) and the rest of the ports in all the 3 Registers are the same as that of the Core Control Registers. The output port of each Output Control Register (1/3 bits) is connected to the write-enable port of the corresponding Output FIFO and the output port of the Output Switch is connected to the read-enable ports of all the Output FIFOs.

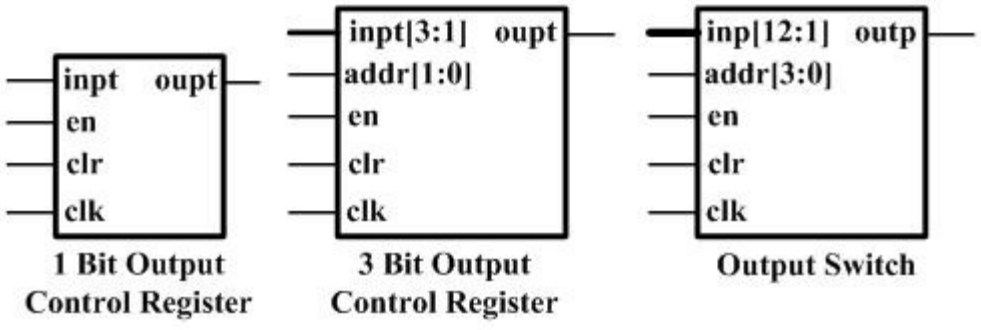

Figure 8.19: Output Control Registers (1/3 bits) & Output Switch

The Figure 8.20 shows the connection diagram of the Output Control Registers along with the Output Switching Circuit. The Output Control registers are connected to the

write-enable port of the Output Data FIFOs and the Output Switch is connected to the read-enable port of all the Output Data FIFOs. Given below are the key points that govern the switching procedure of these Output Control Registers:

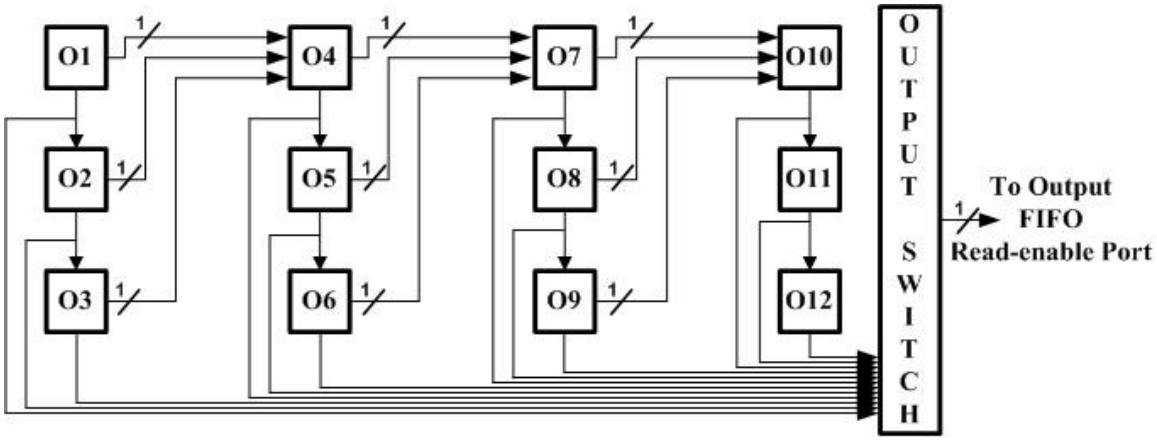

Figure 8.20: Output Control Registers Connection Diagram

- For *I[4,3]*, the output control bit is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*O4*' → '*O5*' → '*O6*' → '*O7*' → '*O8*' → '*O9*' → '*O10*' → '*O11*' → '*O12*' → '*OS*' with one clock cycle delay between each Control Register. '*OS*' represents the Out Switch.
- For *I[4,2]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O4*' → '*O5*' → '*O7*' → '*O8*' → '*O10*' → '*O11*' → '*OS*'.
- For *I[4,1]*, the control data is transferred from '*CC*' → '*O1*' → '*O4*' → '*O7*' → '*O10*' → '*OS*'.
- For *I[3,3]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*O4*' → '*O5*' → '*O6*' → '*O7*' → '*O8*' → '*O9*' → '*OS*'.
- For *I[3,2]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O4*' → '*O5*' → '*O7*' → '*O8*' → '*OS*'.
- For *I[3,1]*, the control data is transferred from '*CC*' → '*O1*' → '*O4*' → '*O7*' → '*OS*'.

- For *I[2,3]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*O4*' → '*O5*' → '*O6*' → '*OS*'.
- For *I[2,2]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O4*' → '*O5*' → '*OS*'.
- For *I[2,1]*, the control data is transferred from '*CC*' → '*O1*' → '*O4*' → '*OS*'.
- For *I[1,3]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*OS*'.
- For *I[1,2]*, the control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*OS*'.
- For *I[1,1]*, the control data is transferred from '*CC*' → '*O1*' → '*OS*'.

8.10 Control Registers Switching Circuit

The image configuration is enabled in the Control Registers using the '*en*' pin shown in figures 8.13 and 8.14. Figure 8.21 shows the combinational switching unit used to enable the Control Registers (both Core & Output) where '*CL[4:0]*' is the channel and level input taken from the Parameter Register 2, '*cin_cont[1:0]*' is the output port that has information about the number of channels in the given Input Image, '*en[12:1]*' provides the control bits to the Control Registers (both Core & Output) depending on the channel/level configuration, '*opt_addr[3:0]*' is the output bits that enable the Output Control Register according to the Configuration and '*sel_addr[1:0]*' is the output port that is connected to the '*addr[1:0]*' port of 3 Data Input Control Register shown in Figure 8.14. Truth tables are used to design the combinational circuit to produce the outputs '*cin_cont[1:0]*', '*en[12:1]*', '*opt_addr[3:0]*', '*sel_addr[1:0]*' from the Channel/Level input from the Parameter Register 2. Truth table for '*en[12:1]*', '*cin_cont[1:0]*', '*opt_addr[3:0]*' and '*sel_addr[1:0]*' are shown in Table 8.3. '*C*' represents the number of channels and '*L*' represents the number of levels in the original image. The circuit is fully combinational and is designed using Gate Level implementation for minimum latency.

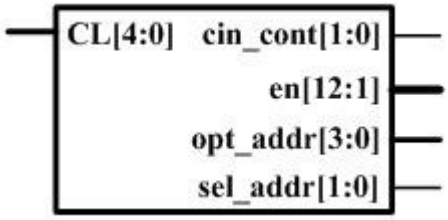

Figure 8.21: Switching Unit for
Core & Output Control Registers

Table 8.3: Truth Table for Control Registers Switching Circuit

C	L	en[12:1]	cin_cont[1:0]	opt_addr[3:0]	sel_addr[1:0]
100 (4)	11 (3)	'111111111111'	'11'	'1100'	'11'
100 (4)	10 (2)	'011011011011'	'11'	'1011'	'10'
100 (4)	01 (1)	'001001001001'	'11'	'1010'	'01'
011 (3)	11 (3)	'000111111111'	'10'	'1001'	'11'
011 (3)	10 (2)	'000011011011'	'10'	'1000'	'10'
011 (3)	01 (1)	'000001001001'	'10'	'0111'	'01'
010 (2)	11 (3)	'000000111111'	'01'	'0110'	'11'
010 (2)	10 (2)	'000000011011'	'01'	'0101'	'10'
010 (2)	01 (1)	'000000001001'	'01'	'0100'	'01'
001 (1)	11 (3)	'000000000111'	'00'	'0011'	'11'
001 (1)	10 (2)	'000000000011'	'00'	'0010'	'10'
001 (1)	01 (1)	'000000000001'	'00'	'0001'	'01'

8.11 **Auto-Write Data Core FIFO**

This section deals with the Input Data Memory control where initially the Data Core FIFO is filled (write operation) with the help of the Input Memory Controller and as the Processor Core starts processing the input data, the Data Core FIFO should be filled accordingly as there exists a delay between each Processor Core. Figure 8.22 shows the Auto-Write circuit that reads the Input Level FIFO and then writes the data obtained from it to the Core Data FIFO.

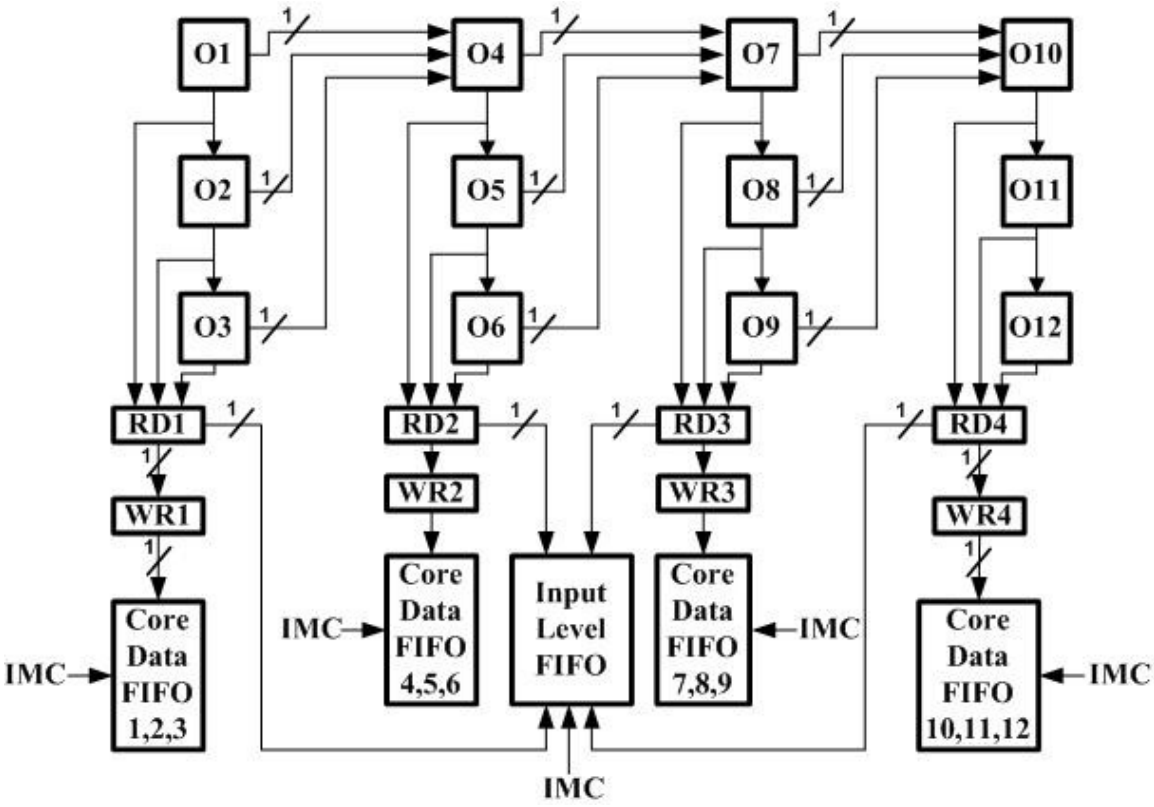

Figure 8.22: Auto-Write Circuit for Core Data FIFO

This circuit uses the control signals from Output Control Registers where '*RD1*', '*RD2*', '*RD3*', '*RD4*' are the read-enable bits managing the read operations of the Input Level FIFO, '*WR1*' is the write-enable bit for Data Core FIFOs (1, 2, 3), '*WR2*' is the write-enable bit for Data Core FIFOs (4, 5, 6), '*WR3*' is the write-enable bit for Data Core FIFOs (7, 8, 9), '*WR4*' is the write-enable bit for Data Core FIFOs (10, 11, 12) and '*IMC*' represents Input Memory Controller unit. '*RD1*', '*RD2*', '*RD3*', '*RD4*' uses the 3-Input Control Register configuration and '*WR1*', '*WR2*', '*WR3*', '*WR4*' uses the 1-Input Control

Register to implement the Auto-Write and Auto-Read operations. The connections to the ports of these registers are similar to the Output Control Registers. Initially the Input Level FIFO is read and the Data Core FIFO is written with the data from the Level FIFO using the Input Memory Controller. This process stops when the Core Data FIFOs are completely full and from this point, the control unit has no control over the write-enable pin of the Data Core FIFOs. The main reason behind this type of implementation is that the input data read from the Core FIFOs does not occur at the same time (different for each Processor Core, 1 clock cycle delay), thus the cores are divided based on channels as shown in Figure 4.13. The image parameters decide the time when the Data Core FIFO should be filled and the procedure is described below briefly.

- For *I[4,3]*, the read and write control is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*RD1*' → '*WR1*', '*O4*' → '*O5*' → '*O6*' → '*RD2*' → '*WR2*', '*O7*' → '*O8*' → '*O9*' → '*RD3*' → '*WR3*', '*O10*' → '*O11*' → '*O12*' → '*RD4*' → '*WR4*'.
- For *I[4,2]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*RD1*' → '*WR1*', '*O4*' → '*O5*' → '*RD2*' → '*WR2*', '*O7*' → '*O8*' → '*RD3*' → '*WR3*', '*O10*' → '*O11*' → '*RD4*' → '*WR4*'.
- For *I[4,1]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*RD1*' → '*WR1*', '*O4*' → '*RD2*' → '*WR2*', '*O7*' → '*RD3*' → '*WR3*', '*O10*' → '*RD4*' → '*WR4*'.
- For *I[3,3]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*RD1*' → '*WR1*', '*O4*' → '*O5*' → '*O6*' → '*RD2*' → '*WR2*', '*O7*' → '*O8*' → '*O9*' → '*RD3*' → '*WR3*'.
- For *I[3,2]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*RD1*' → '*WR1*', '*O4*' → '*O5*' → '*RD2*' → '*WR2*', '*O7*' → '*O8*' → '*RD3*' → '*WR3*'.
- For *I[3,1]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*RD1*' → '*WR1*', '*O4*' → '*RD2*' → '*WR2*', '*O7*' → '*RD3*' → '*WR3*'.

- For *I[2,3]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*RD1*' → '*WR1*', '*O4*' → '*O5*' → '*O6*' → '*RD2*' → '*WR2*'.
- For *I[2,2]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*RD1*' → '*WR1*', '*O4*' → '*O5*' → '*RD2*' → '*WR2*'.
- For *I[2,1]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*RD1*' → '*WR1*', '*O4*' → '*RD2*' → '*WR2*'.
- For *I[1,3]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*O3*' → '*RD1*' → '*WR1*'.
- For *I[1,2]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*O2*' → '*RD1*' → '*WR1*'.
- For *I[1,1]*, the read/write control data is transferred from '*CC*' → '*O1*' → '*RD1*' → '*WR1*'.

The digital components discussed in this chapter constitute to form the Control Unit for this Halftoning Architecture. The circuits are described using Verilog and fully tested using ModelSim. One-Hot Encoding technique is used to maximize the performance and minimize the hardware logic. All the control units in this architecture run at 50 MHz which is also the system frequency.

Chapter 9. System Architecture Performance and Functional Analysis and Results

9.1 Overview

This chapter provides detailed information on the system architecture performance and functional analysis and results results obtained from HDL post-implementation simulation. The Halftoning hardware architecture discussed in the previous chapters is tested with the help of a Verilog test fixture which is a testbench written in Verilog HDL. Input image pixels (CMYK) / parameters are passed to the hardware architecture using a testbench file and the outputs were written to a text file to convert the output pixels to an image format usiing Matlab. The following sections discuss the performance and functional analysis of the architecture and results.

9.2 Performance Analysis and Results

The halftoning algorithm described in this research implements all the basic concepts of blue-noise multitoning with error diffusion. In this algorithm, the number of colors and the number of gray levels are taken from the input image. The Droplet Densities Storage ROM of the architecture are filled with the gray level intensities used for dividing the original image into sub-images. Consider the case of the algorithm being implemented in a traditional sequential CPU and that the input image pixels are processed in a serpentine fashion and that the concept of Parallelism is not used in the original software code. It starts with the first color, goes through every piece of code until it reaches the end of the image. The concept of stacking is only between the different levels of a color but not between different colors. Thus it can be said that the color in one pixel need not wait for a different color in the same pixel for processing. But as the algorithm is run on a CPU and due to the lack of parallelism in the code, the execution takes place in an interdependent sequence. The advantages of randomizing the error filter and the reason for using a serpentine scan method were discussed in section 1.1.3 . Thus the average time taken by a

sequentially executing CPU executing the algorithm for an image size of 799 X 1195 pixels is approximately 2.8 seconds without any other process running in the background. The time taken by the CPU is calculated by using a 'C' code that tells the amount of time consumed to execute a code. The image is run on the CPU for 100 times and an average is taken. The time consumed is more than 3 seconds when this code is run along with other background processes in CPU. It is felt the major advantage of this halftoning algorithm is that the output image or the halftoned image obtained has better quality when compared to other halftoning algorithms. The performance and throughput of the algorithm can be maximized by parallelizing the code to the maximum extent possible which was done prior to development of the parallel implementing system architecture presented within this thesis. The algorithm, written in 'C', was decomposed into segments and each segment is thoroughly analyzed for an equivalent hardware circuit implementation. The resulting system architecture as implemented into the FPGA chip is run at 50 MHz where an output is obtained every 8 clock cycles. The number of pixels of an input image that this hardware can process per second is calculated [36] using Equation 9.1. Hence, the hardware can process 6.25 million pixels per second as shown in Equation 9.2.

$$Throughput\,(Pixels/Second) = \frac{\text{System Clock Frequency}}{\text{Number of Clocks per Output}} \tag{9.1}$$

$$Throughput = \frac{(50*10^{6}\,Clocks/Second)}{(8\,Clocks/Pixel)} = 6250000\ \ Pixels/Second \tag{9.2}$$

The algorithm was run on a single sequential CPU for 100 times and the average execution time was calculated. The output results were tested, verified and the related performance was calculated using the Equation 9.5. According to the Equation 9.2, when an image size of 799 X 1195 pixels is fed as input to the system architecture, the entire image will be processed in 0.153 seconds or in just 153 milliseconds as shown in the calculation 9.3.

$$Execution\ \ Time = \frac{(799*1195\,Pixels)}{(6250000\,Pixels/Second)} = 0.1527688\ \ Seconds \tag{9.3}$$

The initial data buffering operations take about 20 microseconds which is constant for an image of any size and the buffer time when added to the execution time gives the total time taken to process the image shown in calculation 9.4.

$$Execution\ Time\ =\ (0.1527688+(20*10^{-6}))\,Seconds\ =\ 0.1527888\ \ Seconds \quad (9.4)$$

Thus, there is a 18X speedup when the same halftoning algorithm is converted into parallel algorithm and executed on the parallel system architecture executed on an FPGA as shown obtained via Equations 9.5 and 9.6.

$$Speed-up\ =\ \frac{\text{Execution time sequential Architecture CPU}}{\text{Execution time FPGA}} \quad (9.5)$$

$$Speed-up\ =\ \frac{2.81677\ \text{Seconds}}{0.1527888\ \text{Seconds}}\ =\ 18.4357 \quad (9.6)$$

The algorithm is designed to handle wide format images and currently the hardware can support images up to a size of 24 X 44 inches which is equal to 548 million pixels. The time taken to process an image of this size on a conventional CPU is about 27 minutes which can also create problems in printing the image. The parallel architecture implemented into a in the FPGA takes only 87.6 seconds (1.46 minutes) to process a 24 X 44 inch size image which shows a large margin of improvement in the performance of the equivalent hardware unit. The total area of the image that can be processed per second is given in the Equations 9.7 and 9.8.

$$Image\ Area\ Processed\,(Square\ Inches/Second)\ =\ \frac{\text{Image Size (Square Inches)}}{\text{Execution time (Seconds)}} \quad (9.7)$$

$$Image\ Area\ Processed\ =\ \frac{(24*44)}{(87.6)}\ =\ 12.054\ \ Square\ Inches/Second \quad (9.8)$$

Figure 9.1 shows the graphical representation of sequential CPU execution time (Seconds) versus the parallel architecture implementation (CPU & FPGA) execution time for a 799 X 1195 image size.

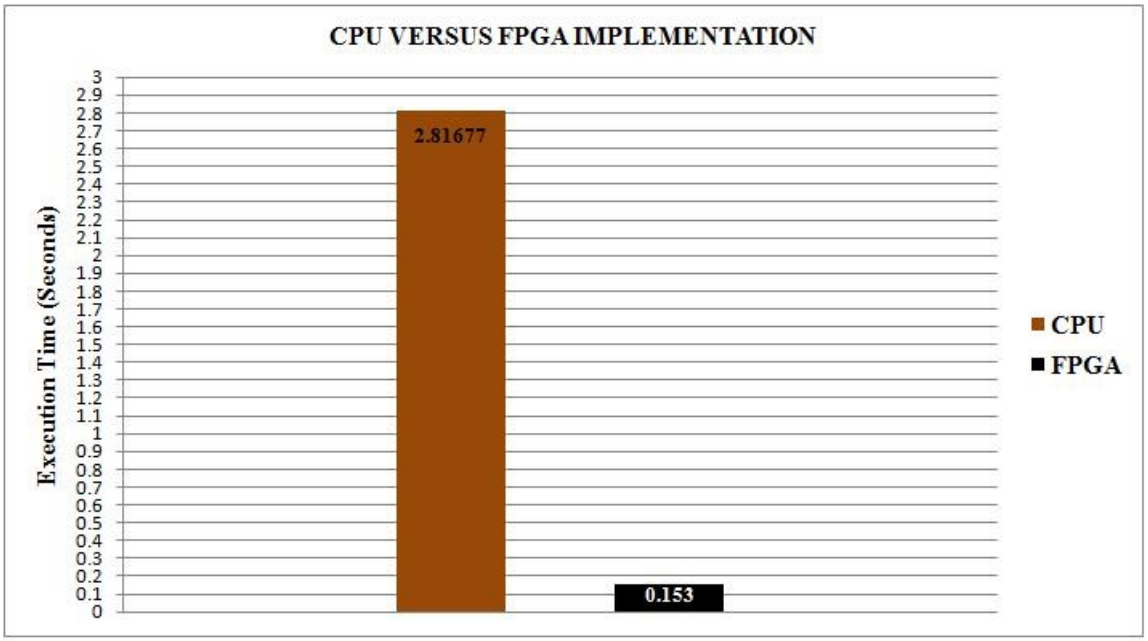

Figure 9.1: Graph Showing Execution Times of a Single CPU and Parallel Halftoning Architecture Implemented to a FPGA

9.3 HDL Functional and Performance Simulation Validation of Parallel Halftoning Architecture

The HDL simulation results were obtained from the Mentor Graphics ModelSim CAD simulation tool software [20]. The simulation results shown in this section are Post-Place and Route HDL simulation results implying all hardware propagation delays are included in the simulation results meaning the simulation results can be used to evaluate the functionality and performance of the parallel architecture. There is no way to show results from each and every functional unit in the architecture. All main components, functional units and the entire parallel architecture are covered and discussed. The architecture uses 13 clocks in total to minimize clock skew where '*clk*' is the clock input to the Input Data Memory Architecture, '*clk1*' through '*clk12*' are used for each Processor Core starting from 1 through 12. Figures 9.2 to 9.28 shows the simulation results of the Parallel Hardware Halftoning Architecture using a step by step approach. The shown results start from buffering the input pixels to the Input Image FIFO, converting the input data to 12 bit data, reading the corresponding droplet densities from the Droplet Densities ROM,

filling up the Level FIFOs with the values from the ROM, filling the Core Data FIFO, starting the Processor Cores when the Data FIFOs are full, calculating the error value, running the Error-Diffusion unit to disperse errors to the neighboring pixels, storing the errors generated in a Block Memory storage unit and finally buffering of the output every 8 clock cycles. The system is run at 50 MHz which is the timing constraint set to the hardware. Each and every component (functional unit) in the Architecture was fully tested and validated. A HDL testbench was written using the Verilog Test Fixture software in the Xilinx ISE 10.3 CAD tool set and the outputs of the testbench were simulated using ModelSim. The input pixels are extracted from the input image using MATLAB and stored in a text file as shown in Figure 9.3. The data in the text file is accessed by the test fixture software and fed to the internal memory of the FPGA. The output from the simulation is directly written to a text file in a binary format as shown in Figure 9.28 and they are converted to an image with the help of a MATLAB code. The obtained results were thoroughly analyzed and validated using the ModelSim and MATLAB CAD tools. Considering all obtained HDL Post-Place and Route simulation results shown in 9.2 through 9.39, it was concluded that the previously presented Special Purpose Parallel Architecture correctly executes the new Stacked Error Diffusion Halftoning Algorithm.

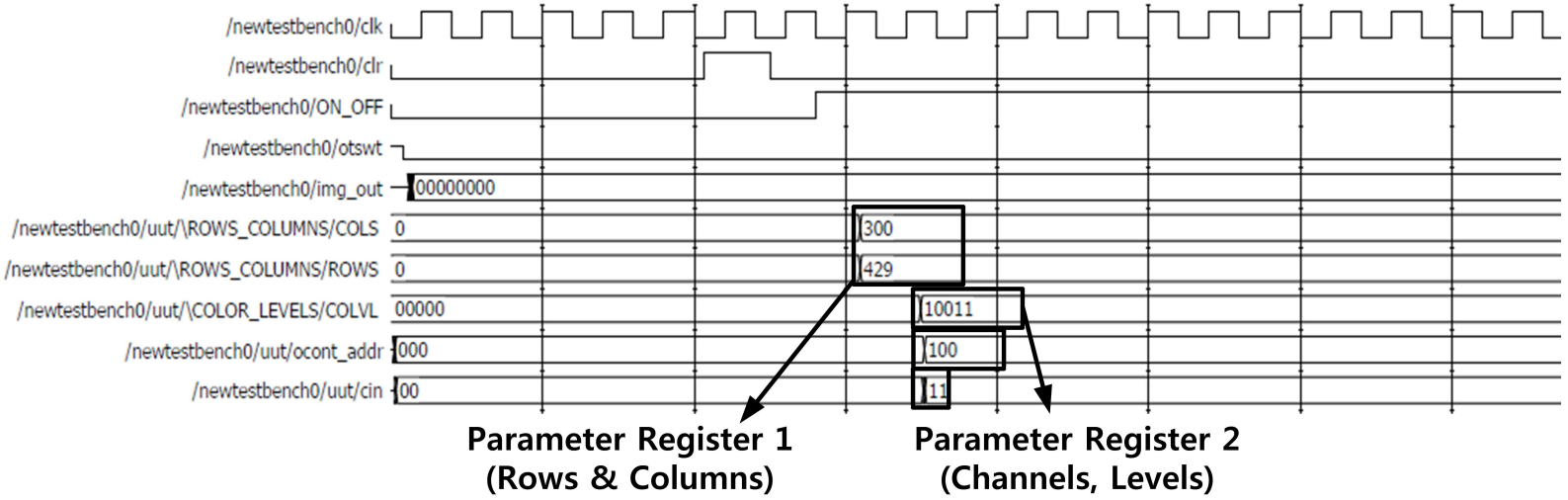

Figure 9.2: Parameter Register 1 & 2 - Simulation Result

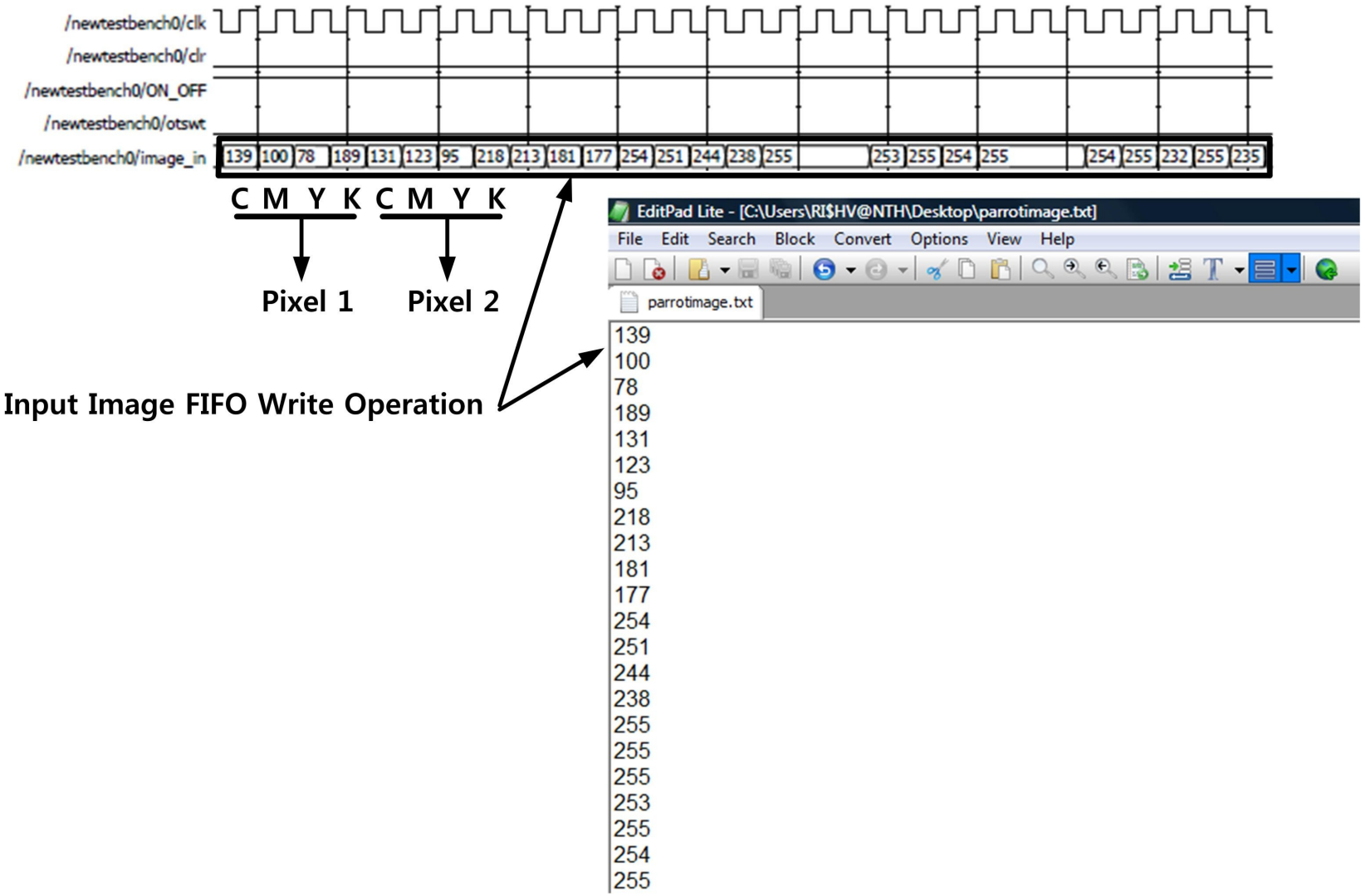

Figure 9.3: Data Buffering Operation in Input Image FIFO - Simulation Result

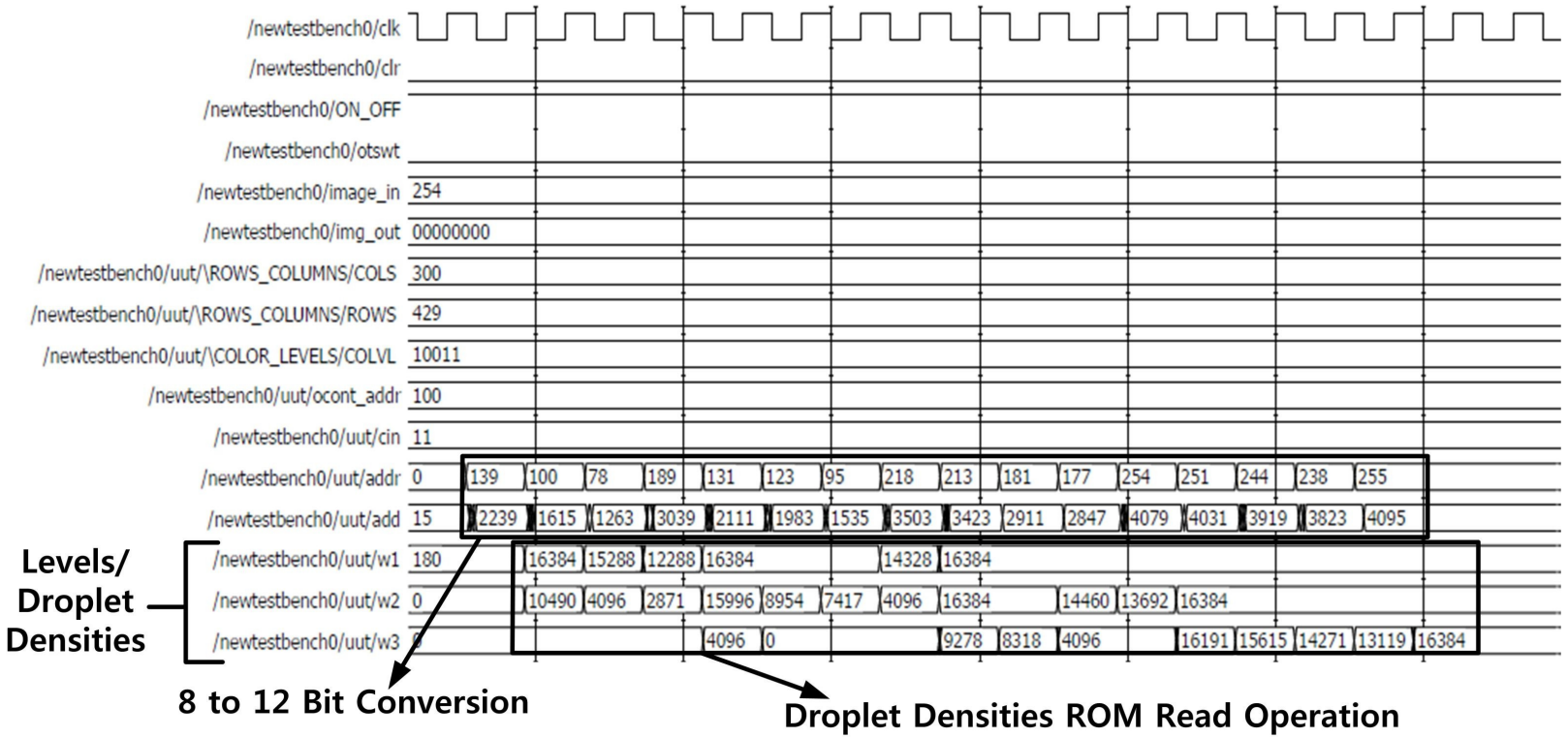

Figure 9.4: 8 to 12 Bit Coversion and Droplet Densities Mapping - Simulation Result

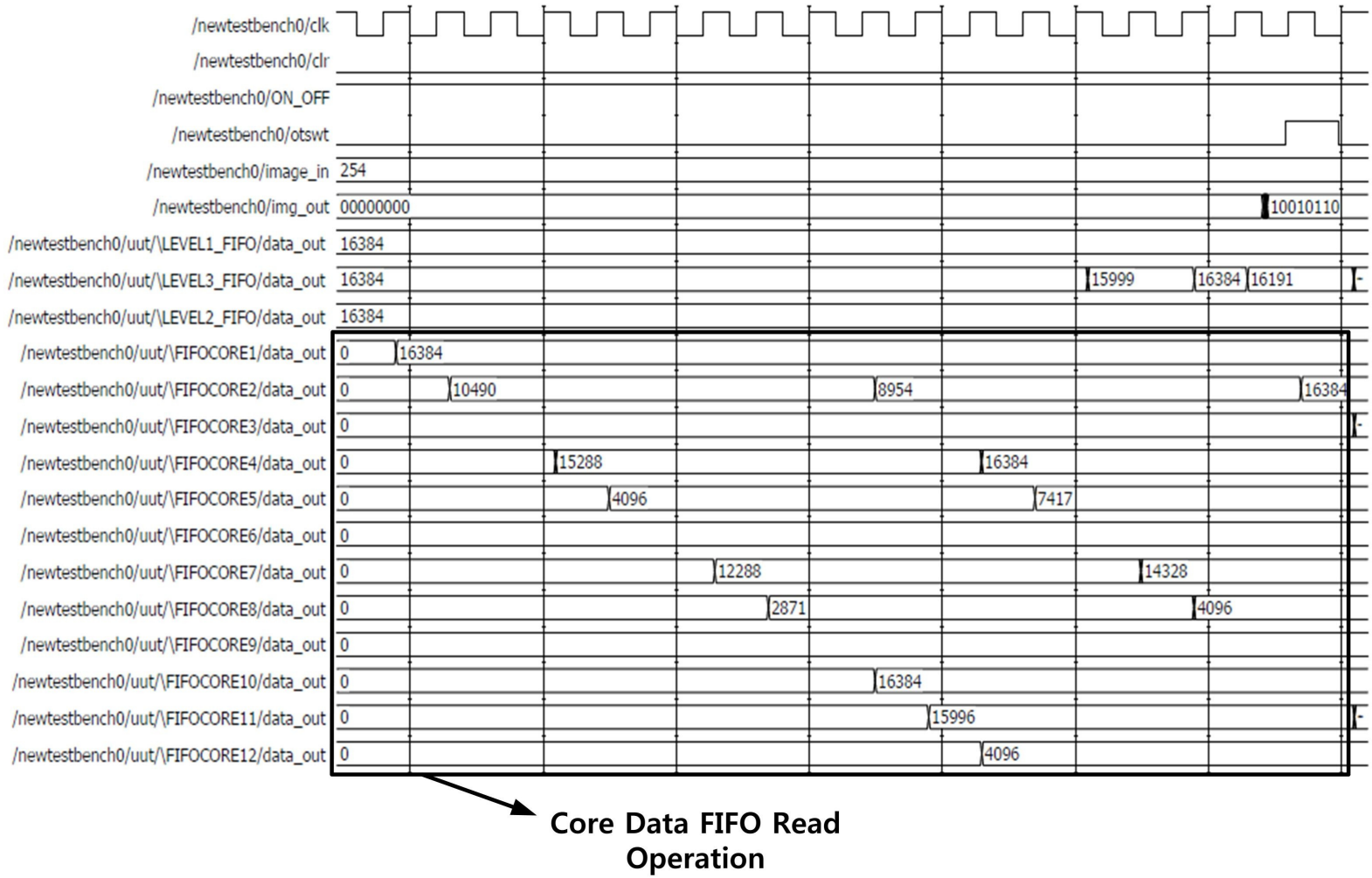

Figure 9.5: Core Data FIFOs [1-12] - Simulation Result

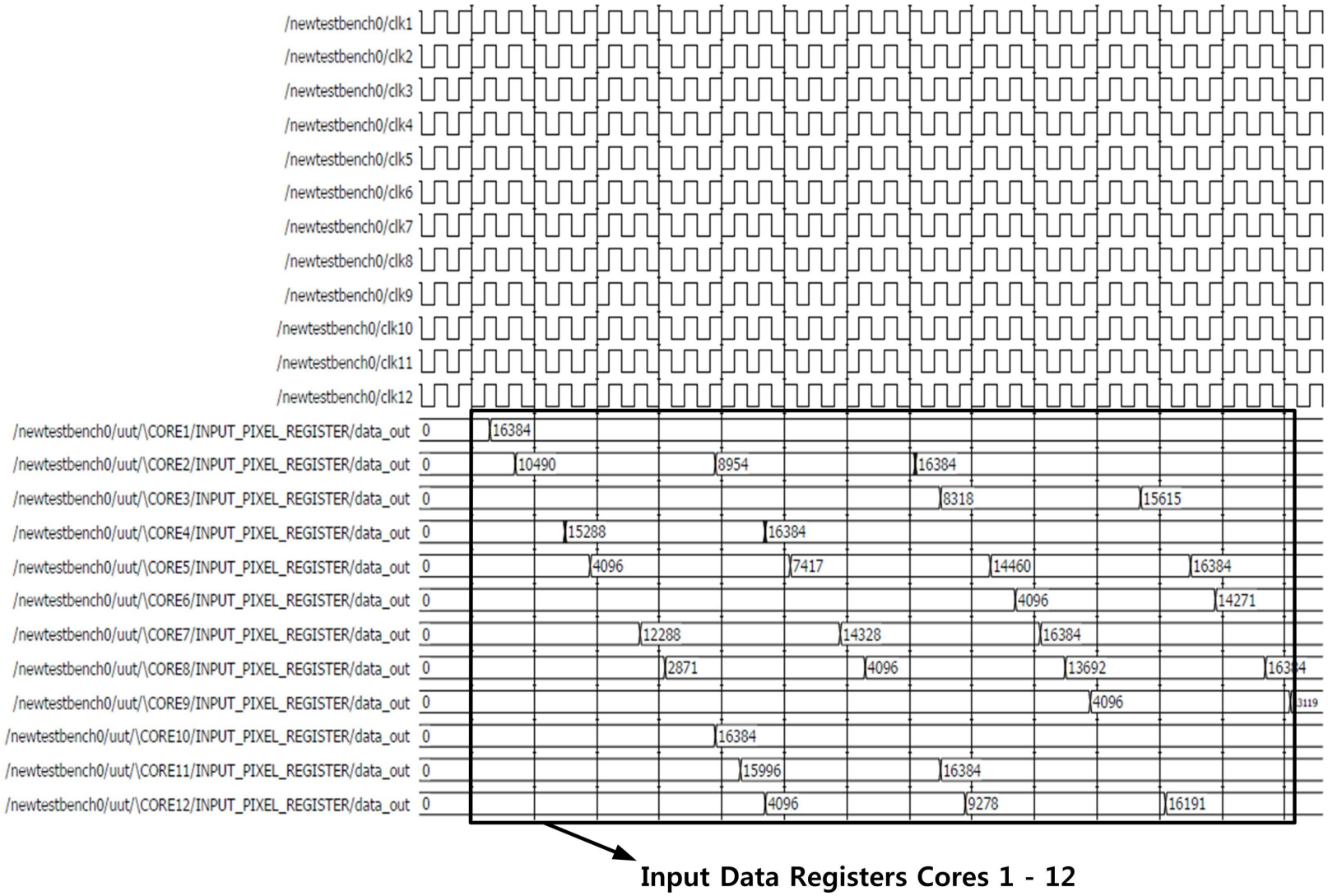

Figure 9.6: Input Pixel Register [1-12] Data Values - Simulation Result

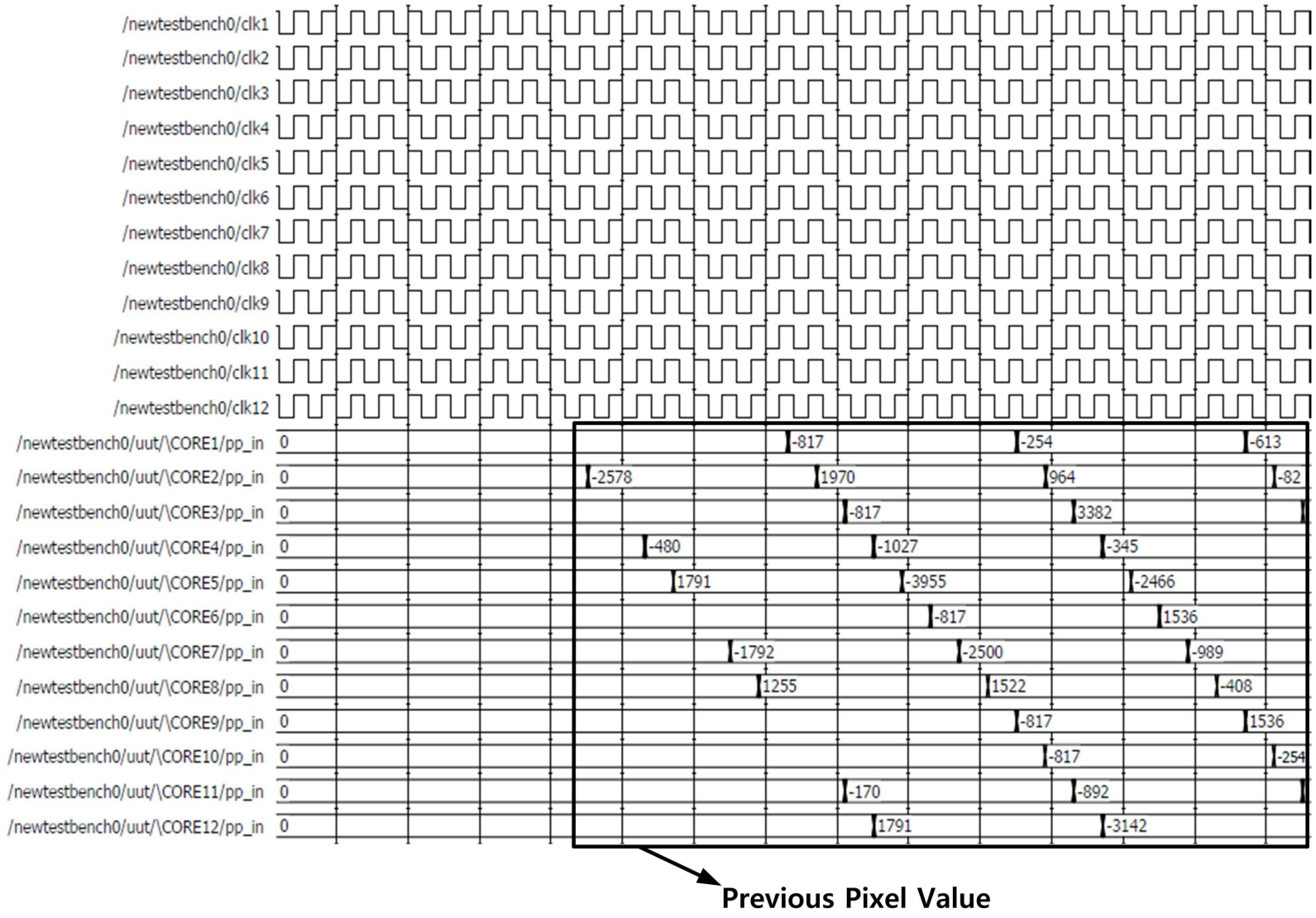

Figure 9.7: Previous Pixel Values [1-12] - Simulation Result

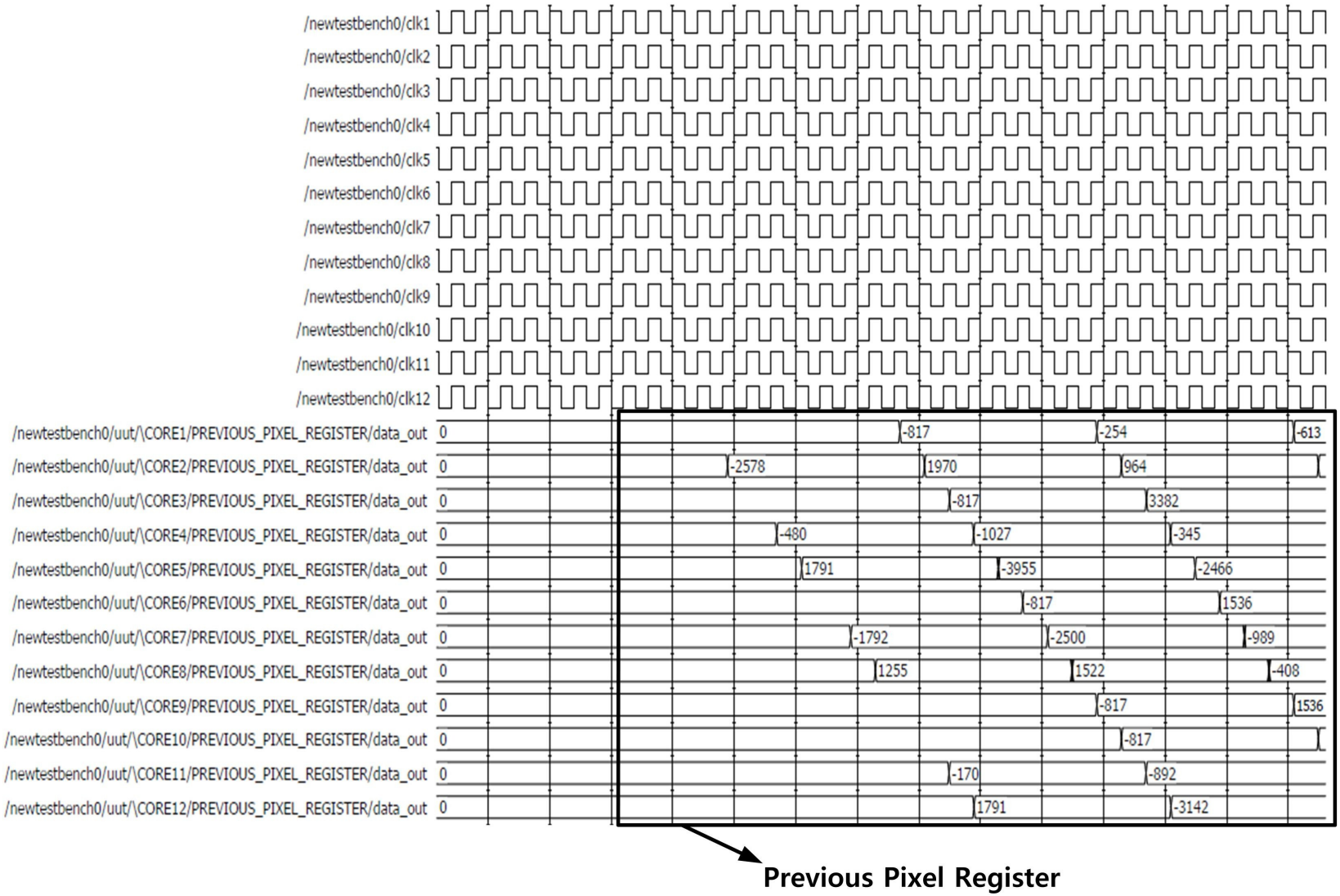

Figure 9.8: Previous Pixel Register [1-12] Data Values - Simulation Result

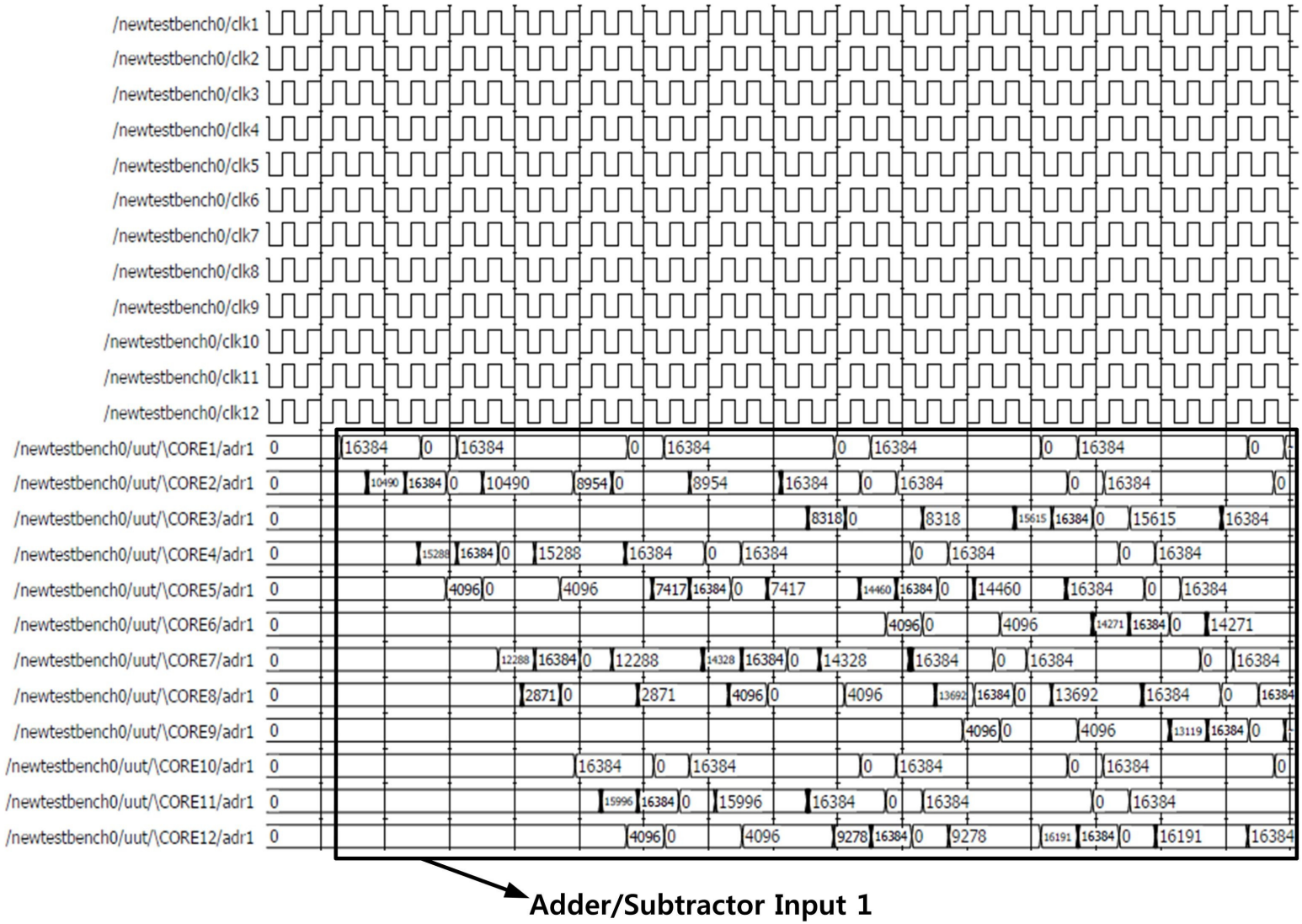

Figure 9.9: Input 1 of Adder-Subtractor Unit [1-12] - Simulation Result

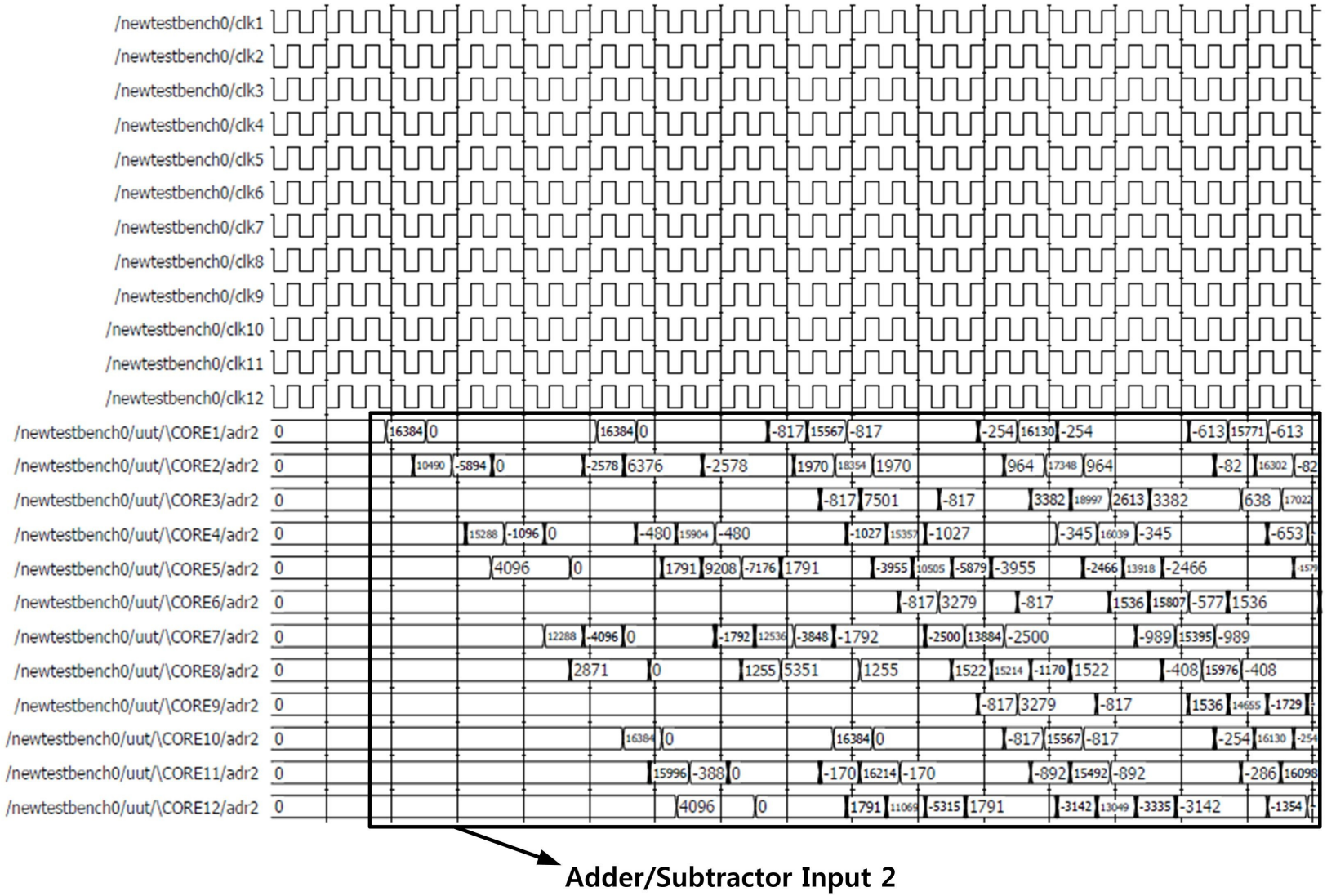

Figure 9.10: Input 2 of Adder-Subtractor Unit [1-12] - Simulation Result

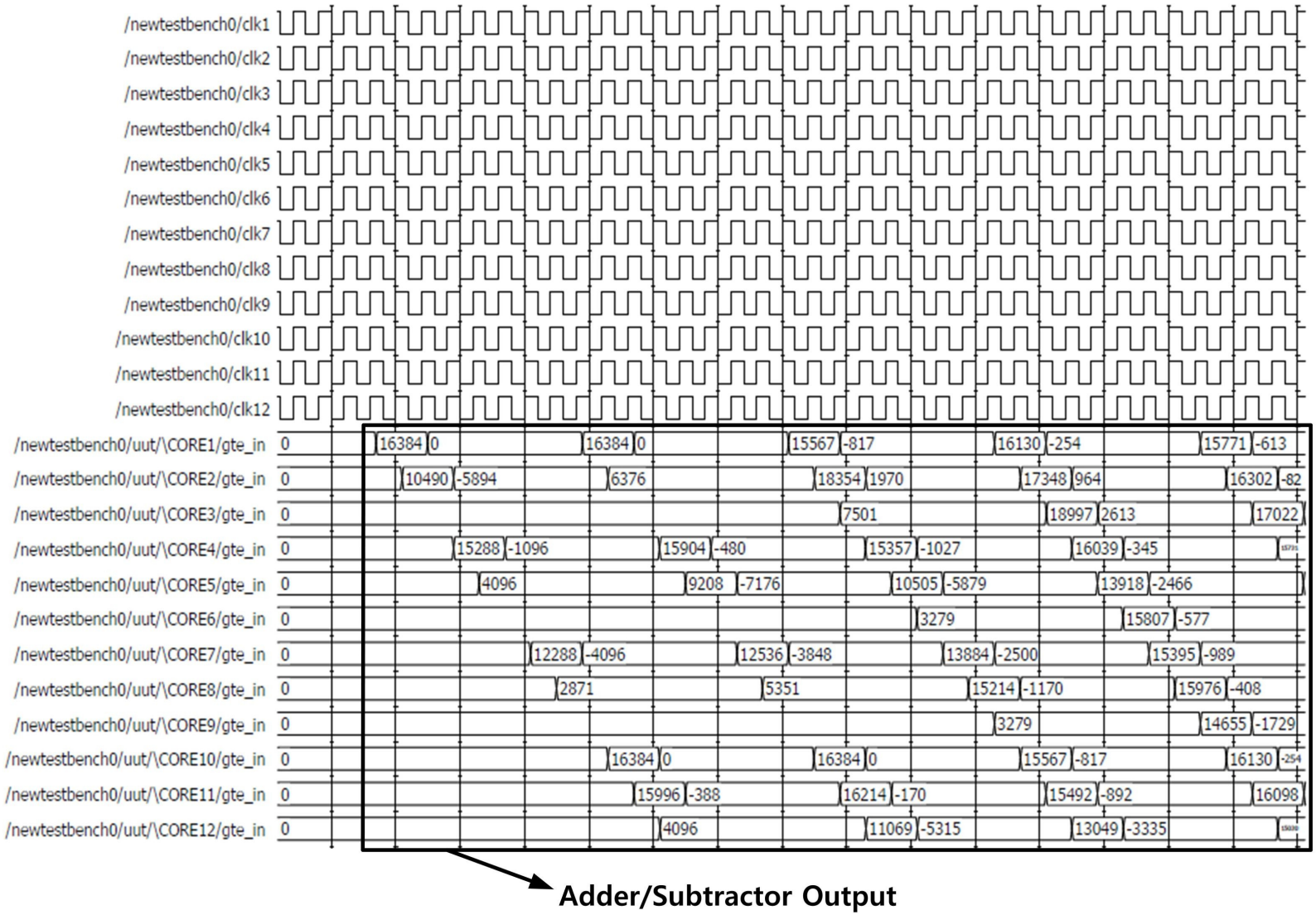

Figure 9.11: Output of Adder-Subtractor Unit [1-12] - Simulation Result

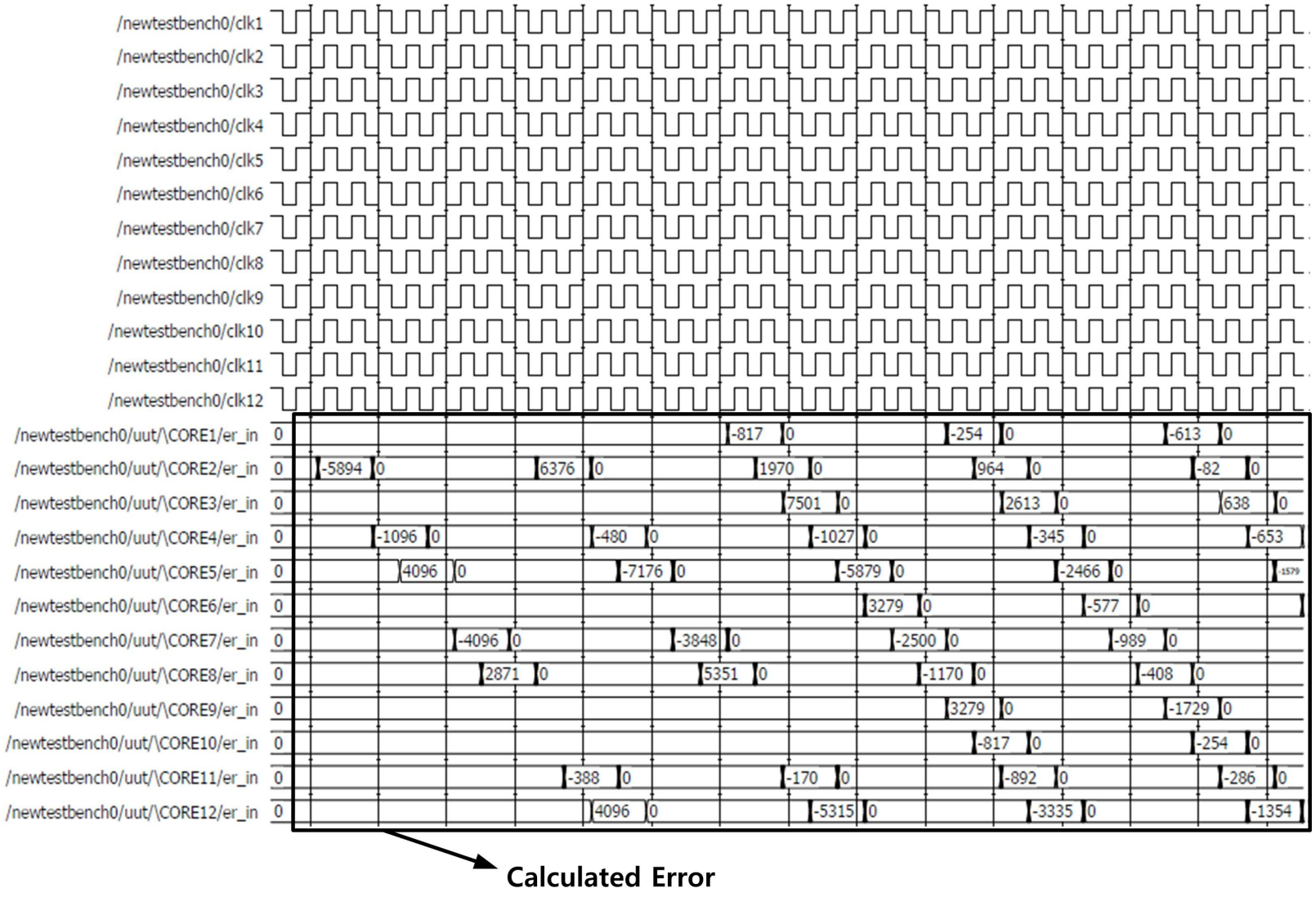

Figure 9.12: Calculated Error Values [1-12] - Simulation Result

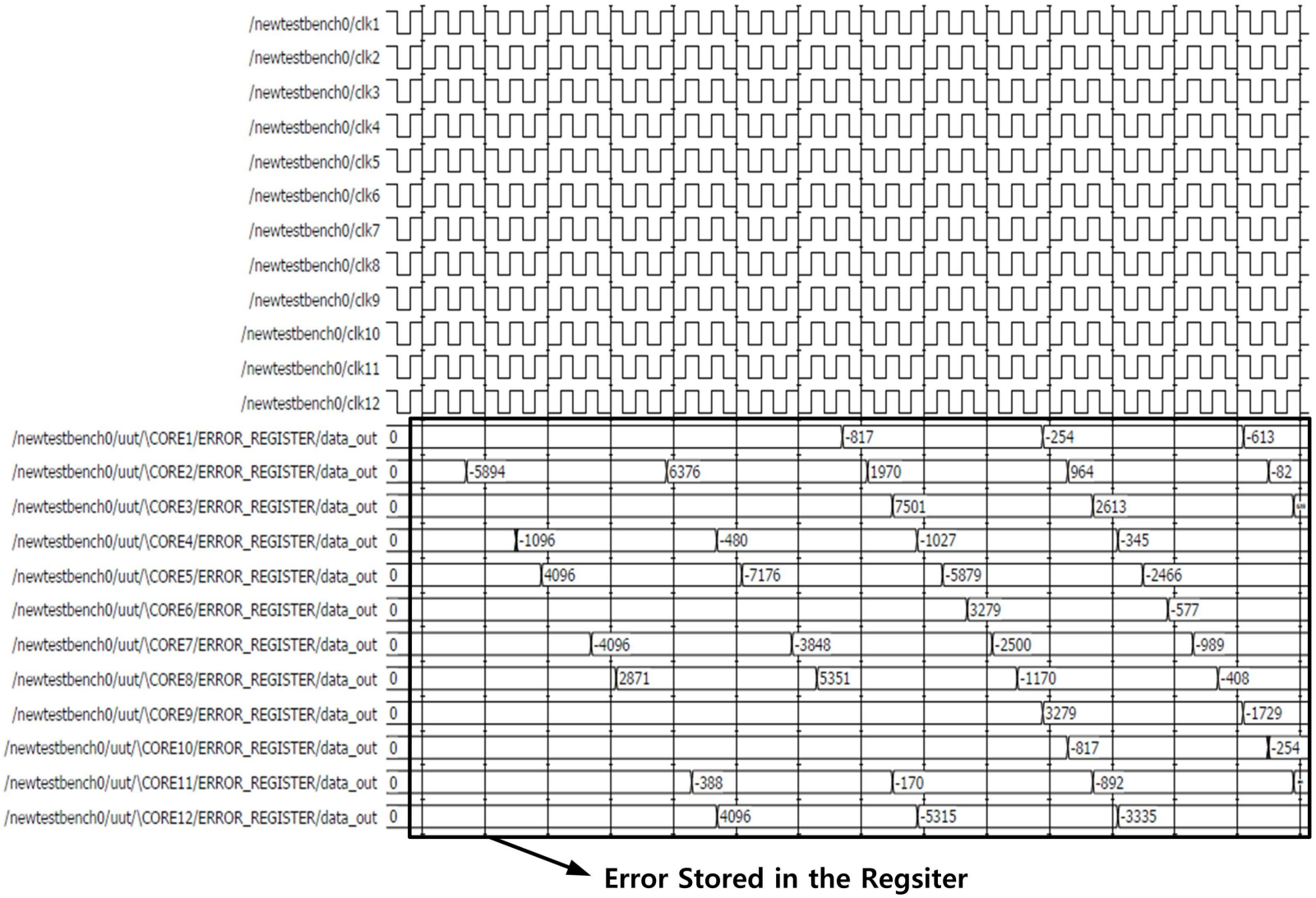

Figure 9.13: Error Values Stored in Error Register [1-12] - Simulation Result

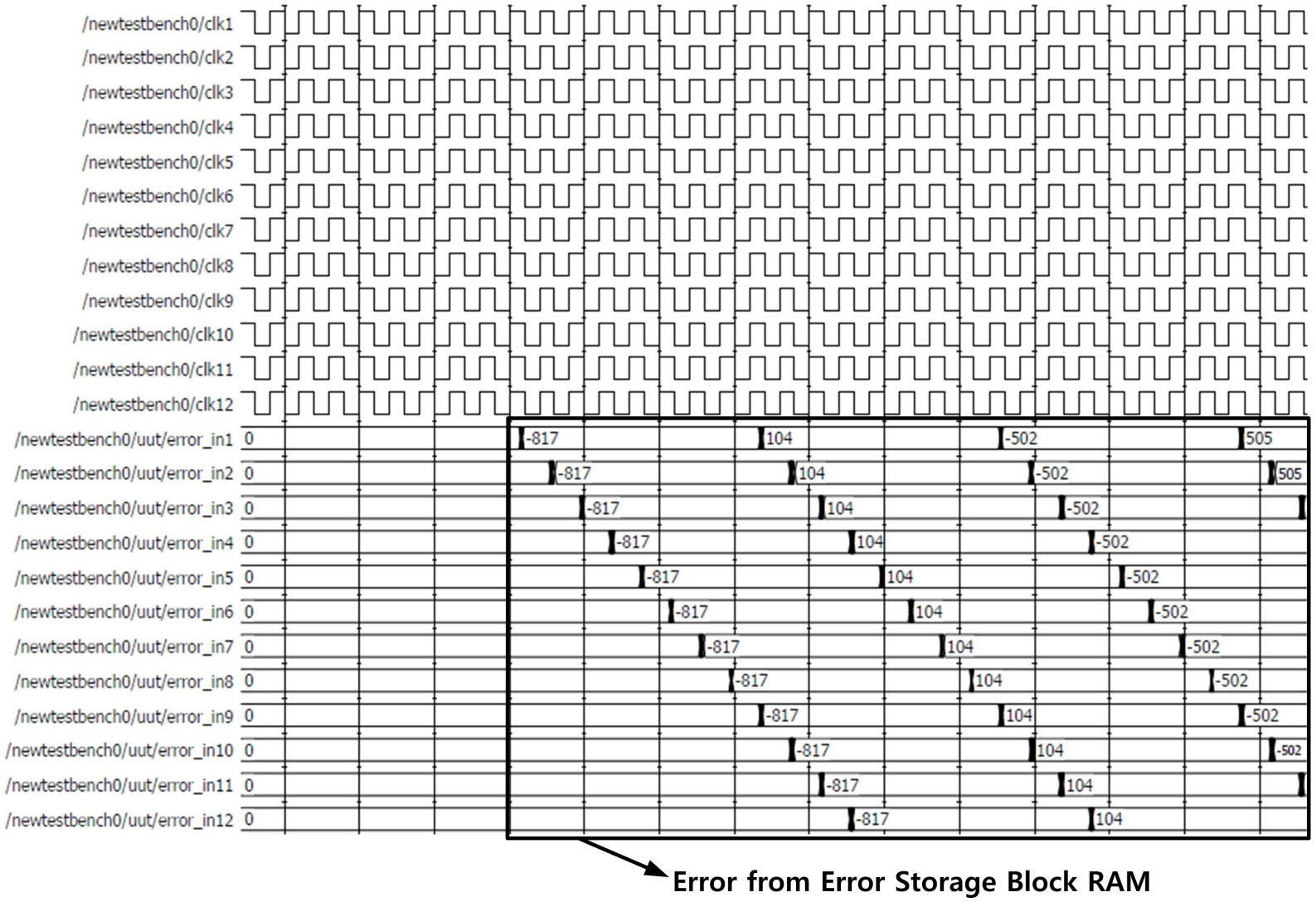

Figure 9.14: Error Values From Error Storage Block RAMs [1-12] - Simulation Result

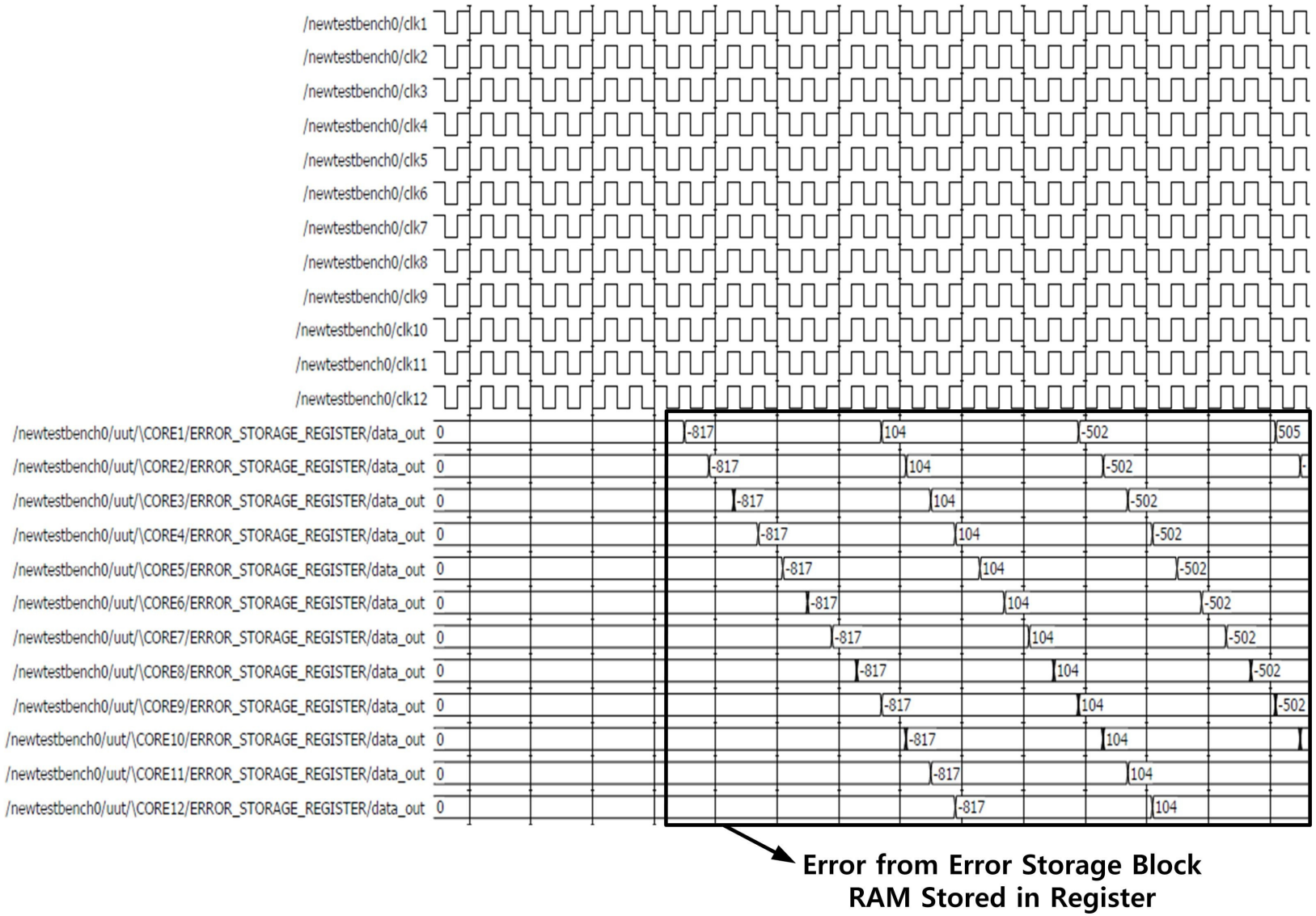

Figure 9.15: Error Values Stored in Error Storage Registers [1-12] - Simulation Result

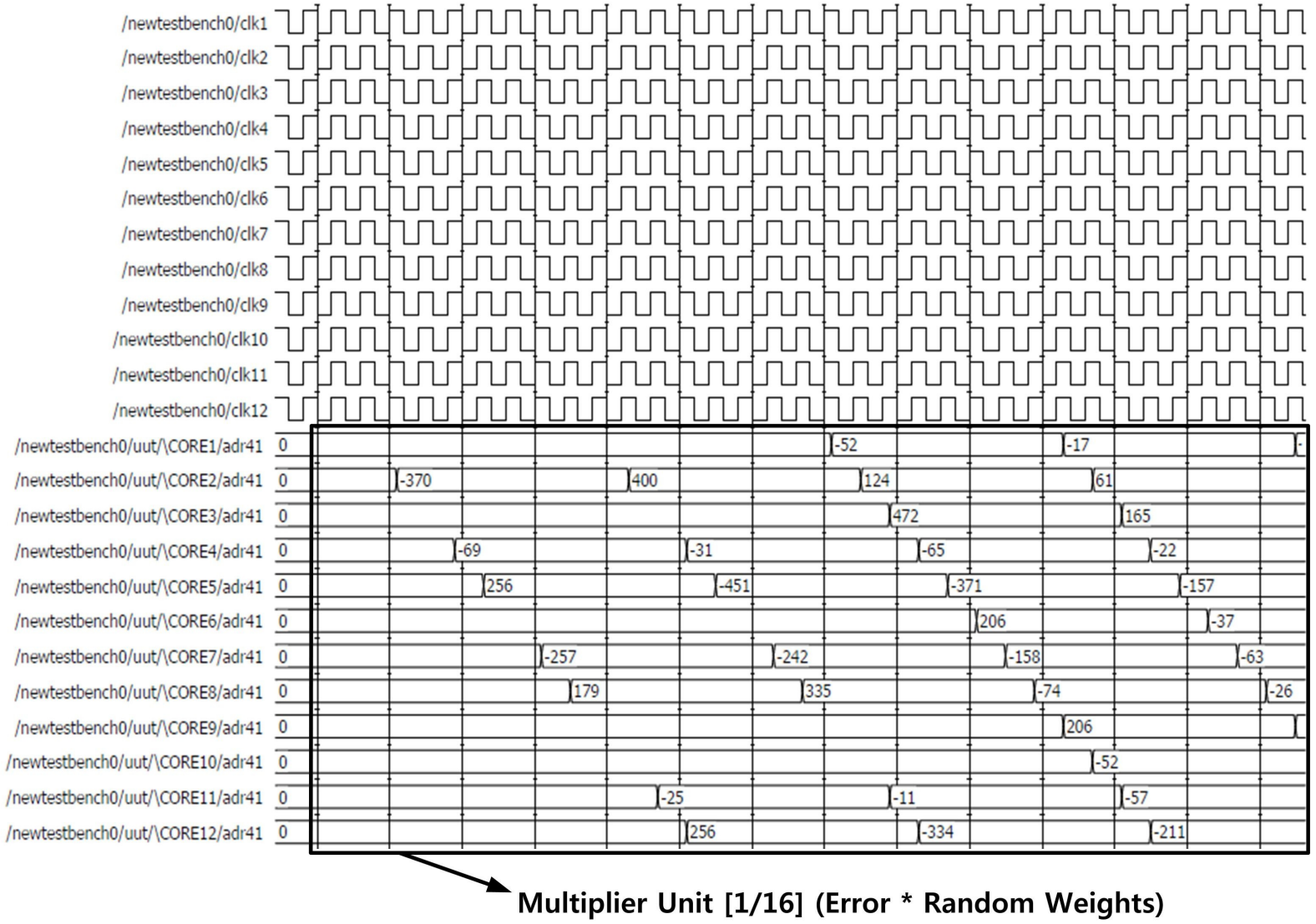

Figure 9.16: Output of Multiplier Unit [1/16] - [1-12] - Simulation Result

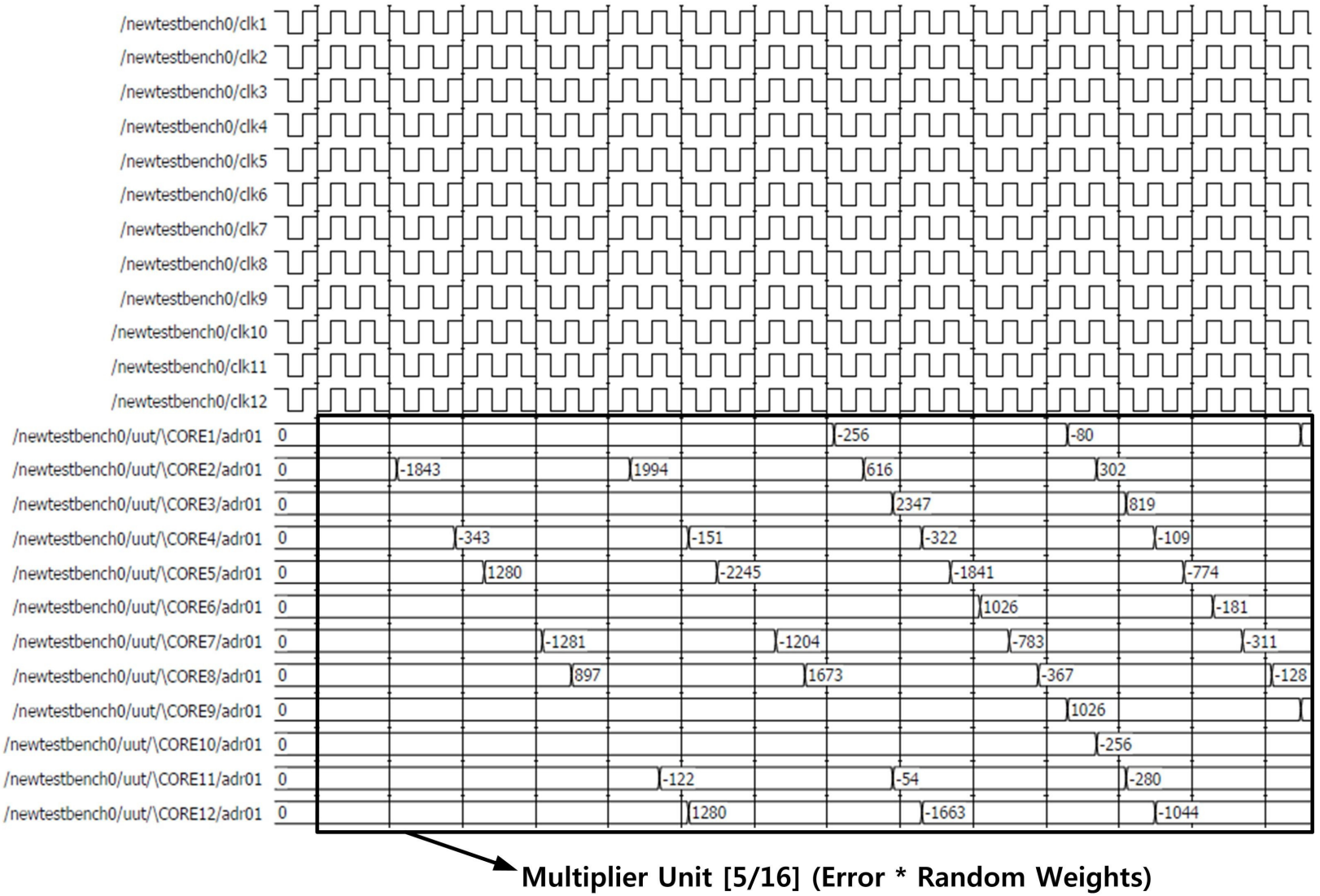

Figure 9.17: Output of Multiplier Unit [5/16] - [1-12] - Simulation Result

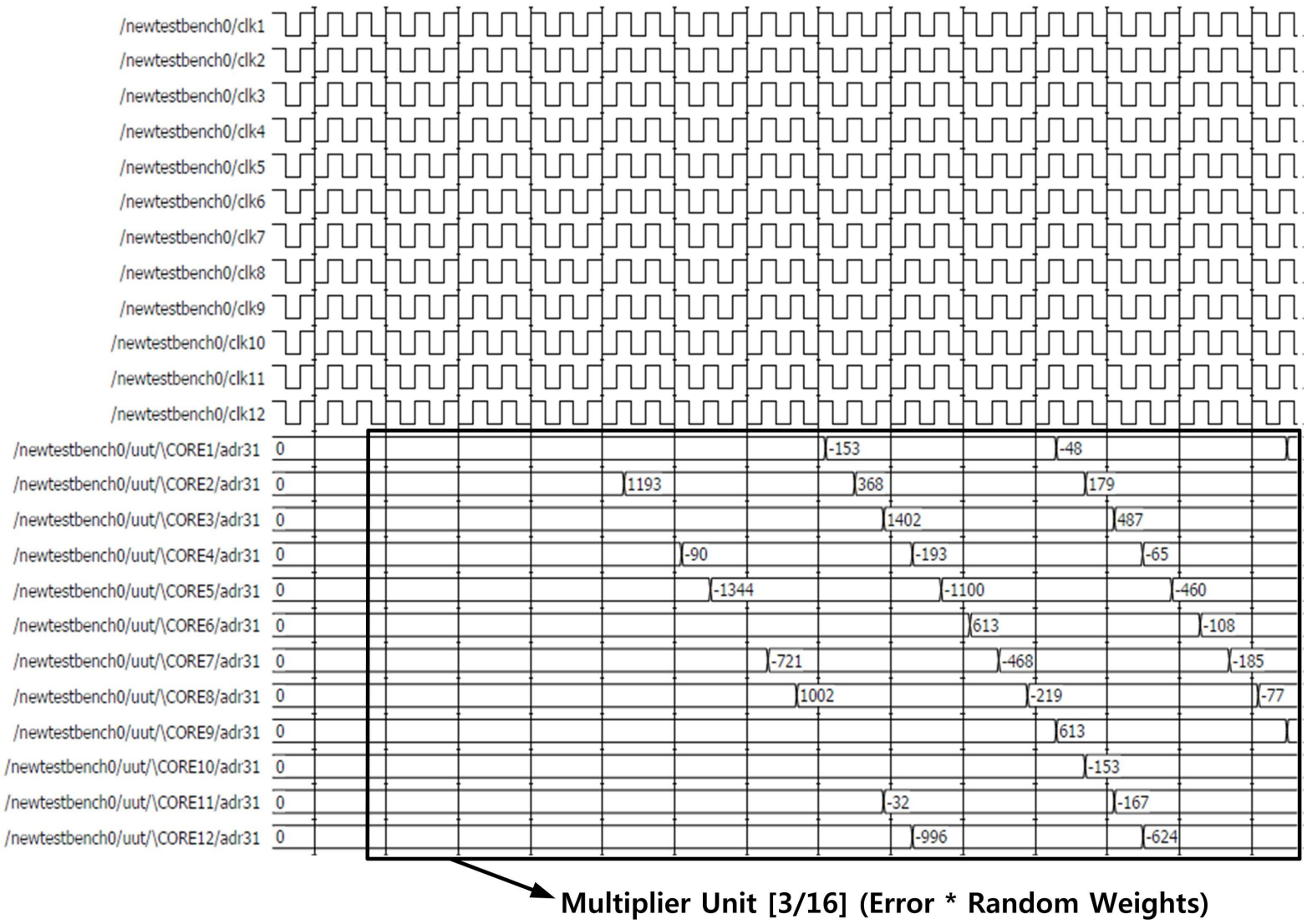

Figure 9.18: Output of Multiplier Unit [3/16] - [1-12] - Simulation Result

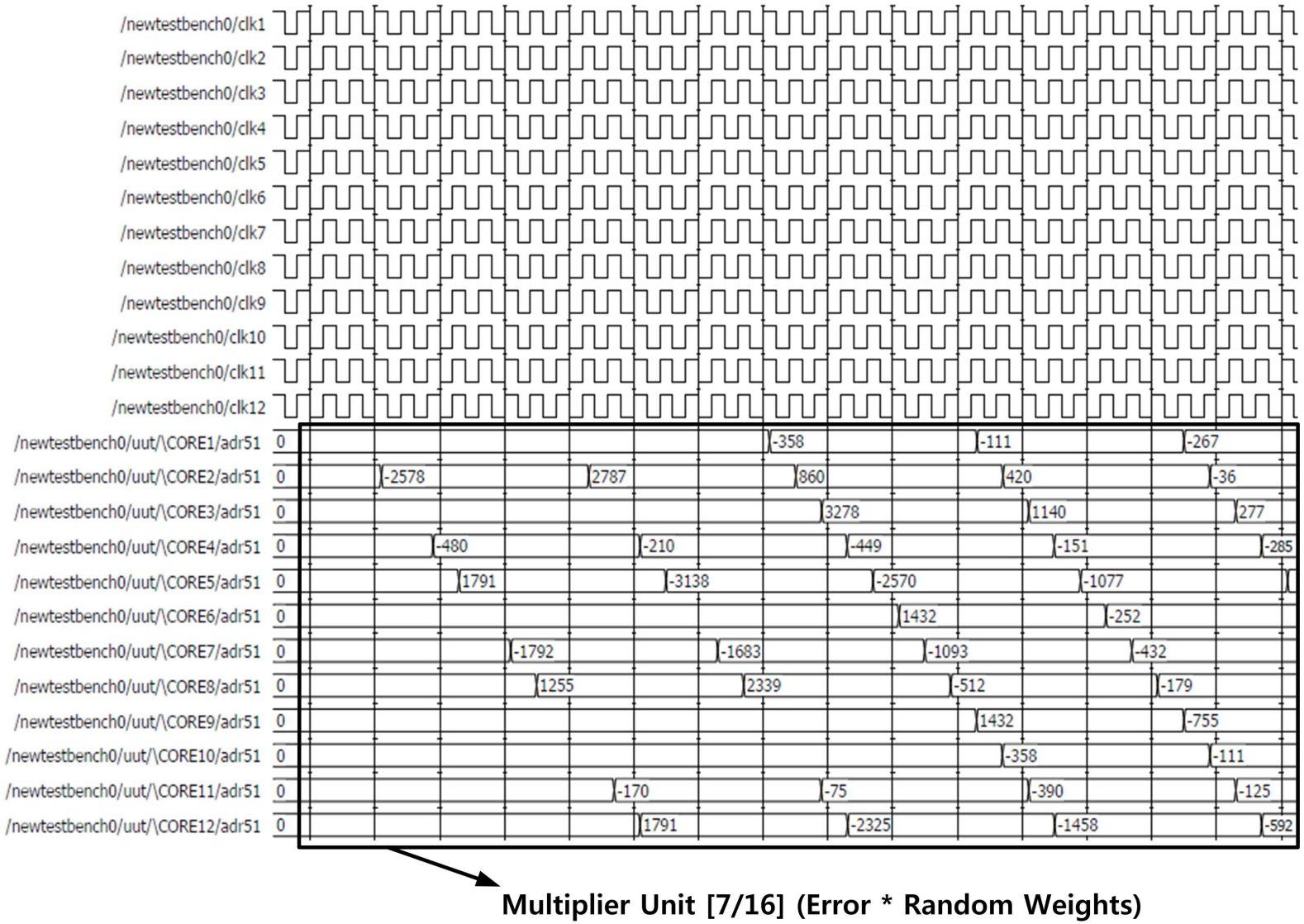

Figure 9.19: Output of Multiplier Unit [7/16] - [1-12] - Simulation Result

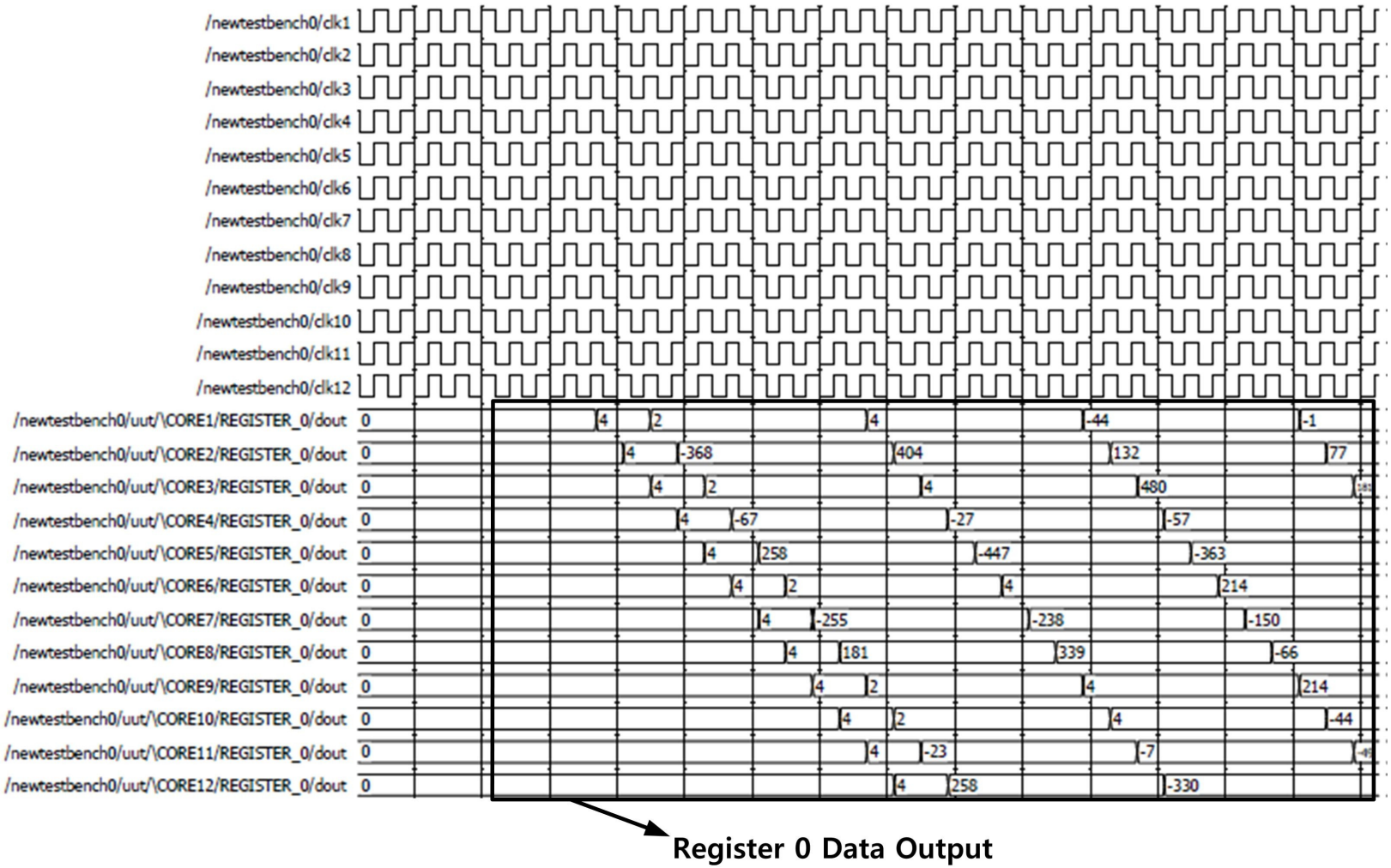

Figure 9.20: Data Output From Register [5/16] - [1-12] - Simulation Result

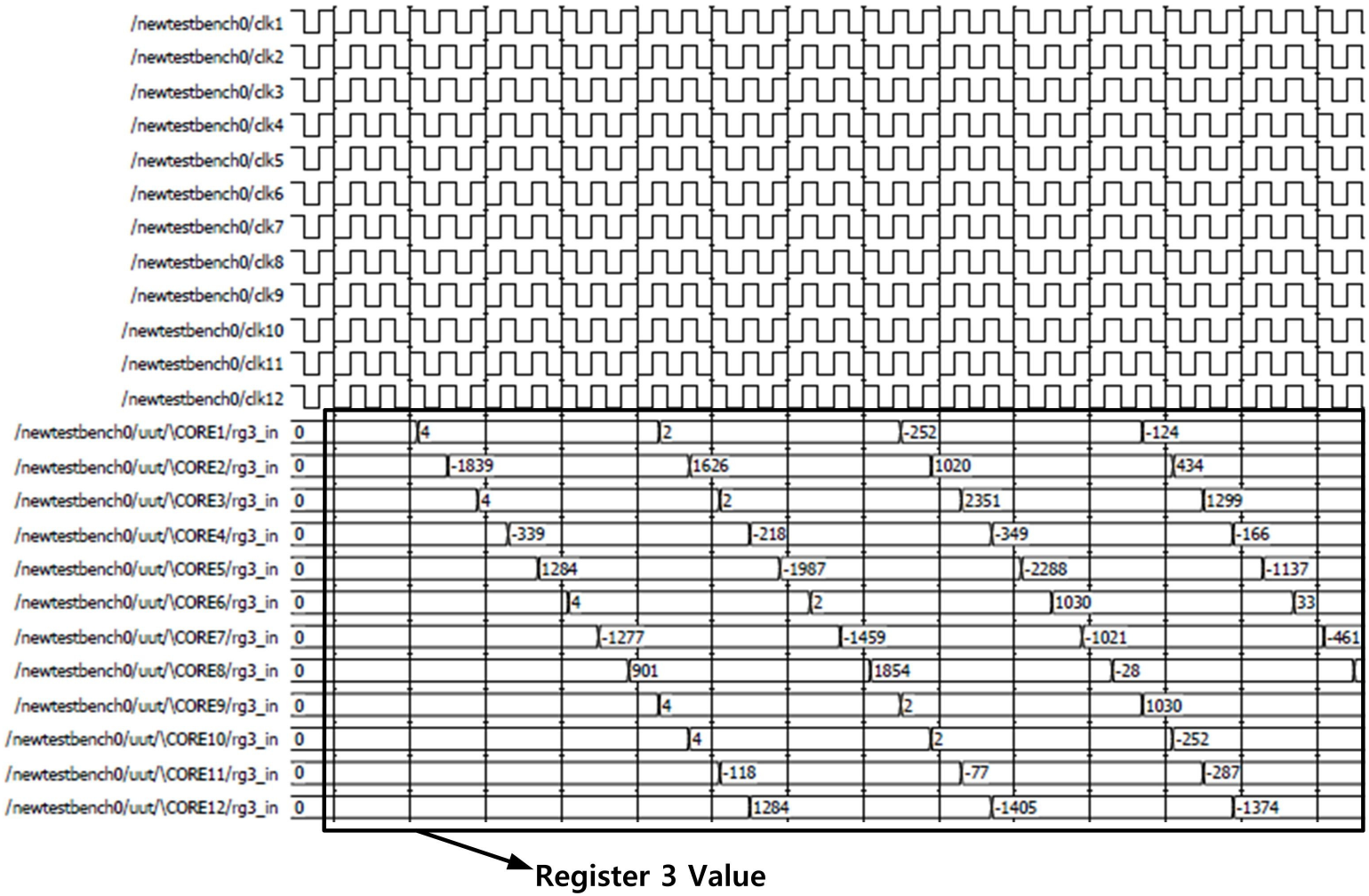

Figure 9.21: Data Output From Register [3/16] - [1-12] - Simulation Result

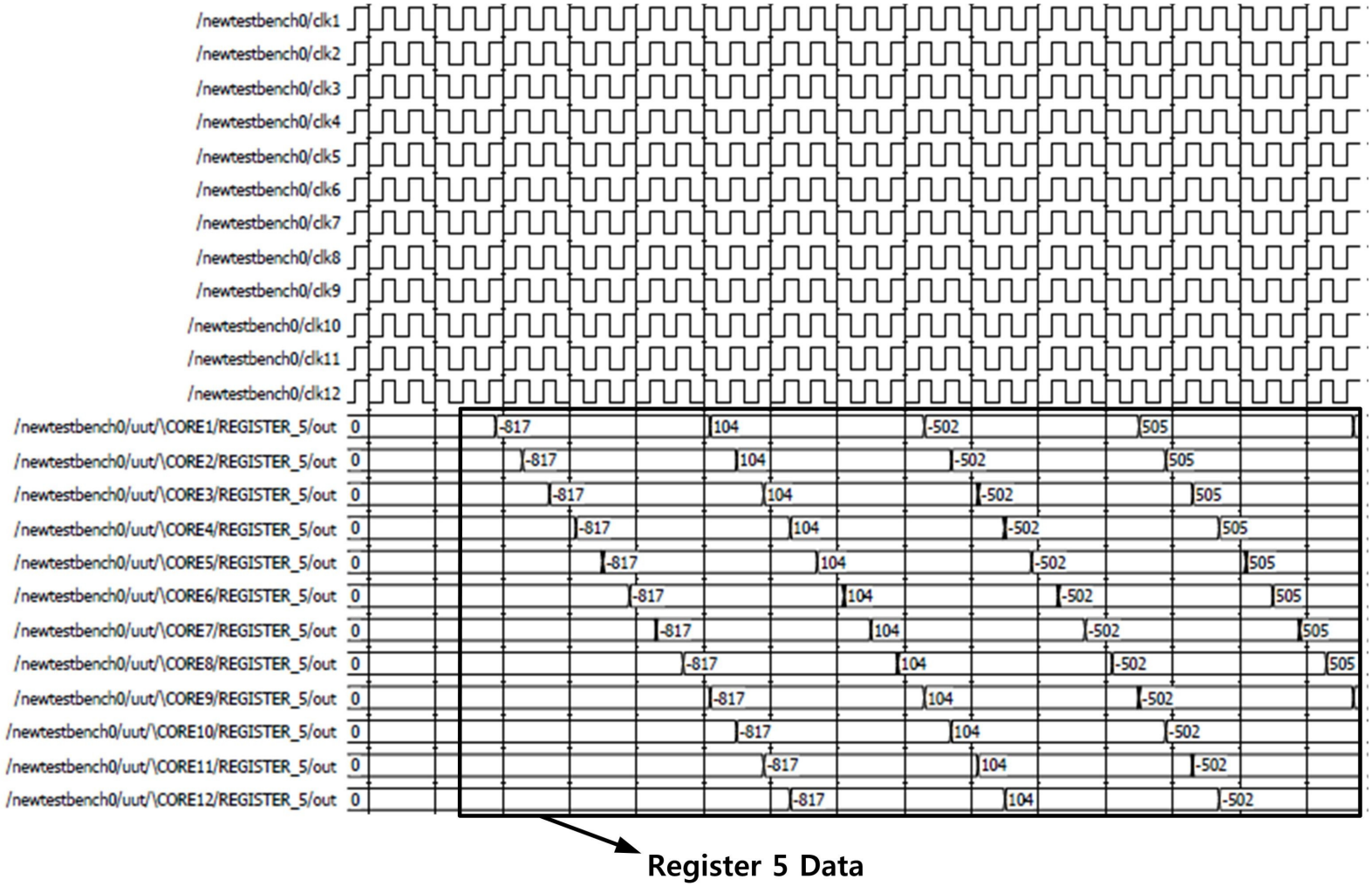

Figure 9.22: Data Output From Register [7/16] - [1-12] - Simulation Result

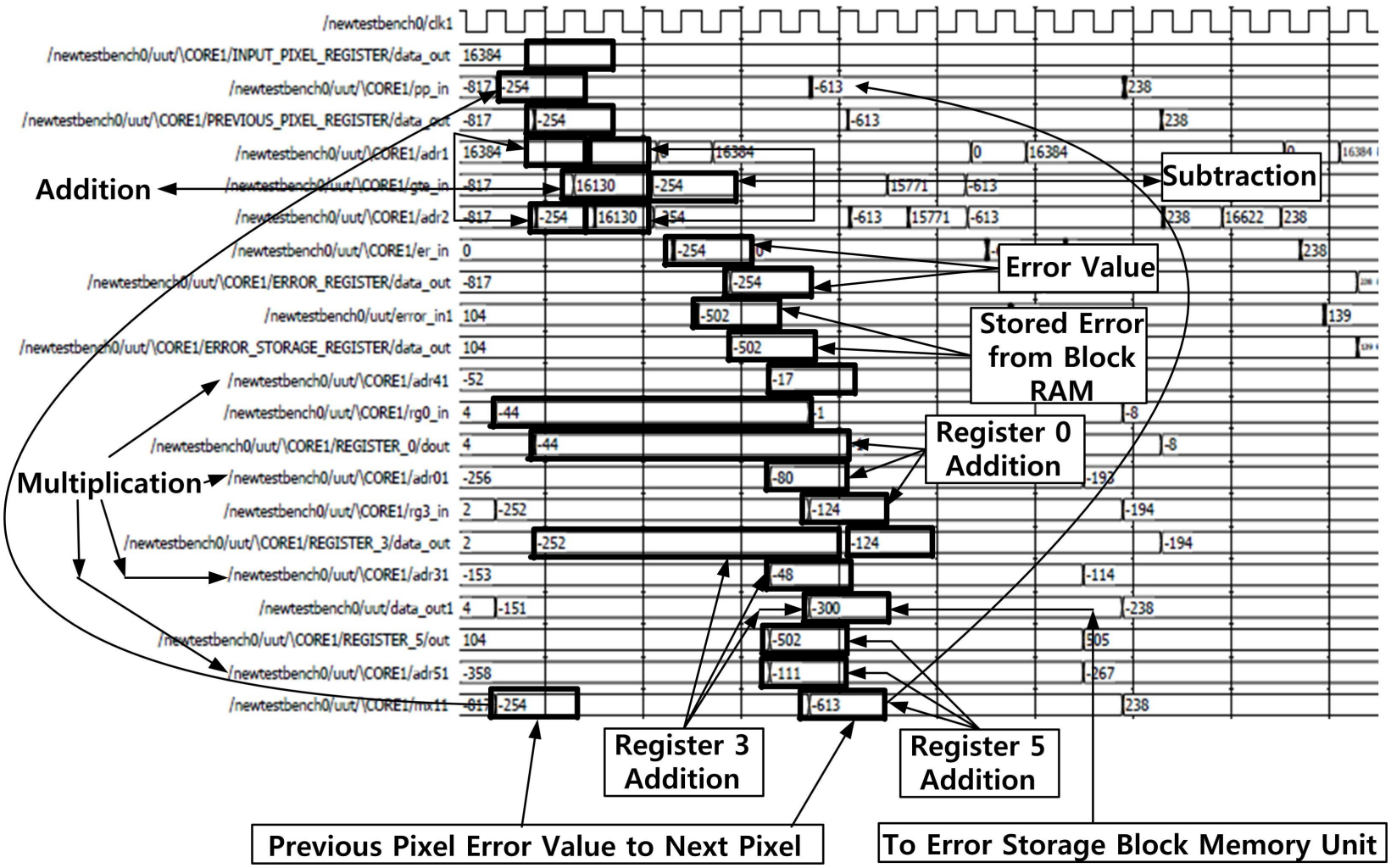

Figure 9.23: Processor Core 1 Data Operations - Simulation Result

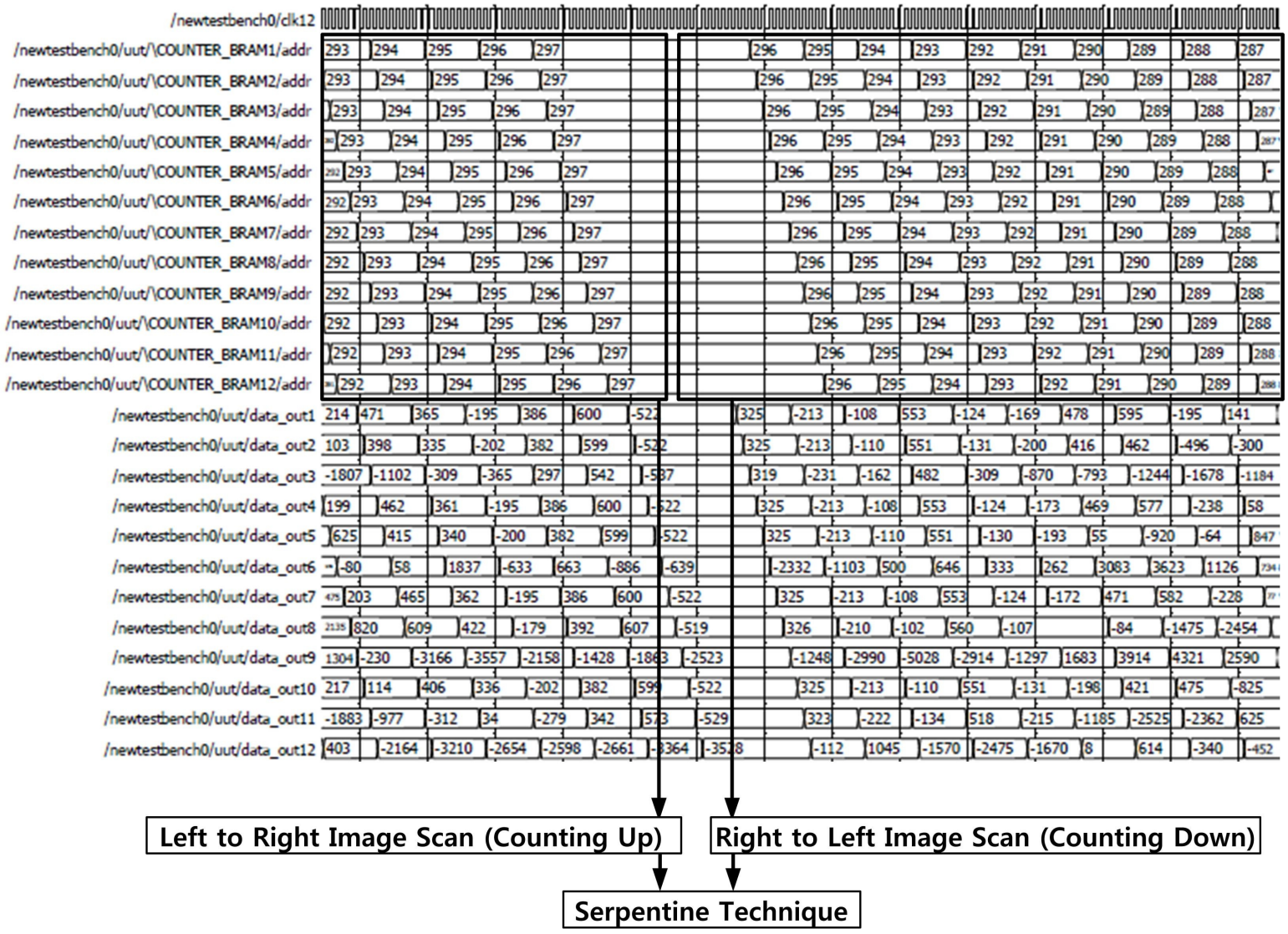

Figure 9.24: Error Storage Block RAM Address Counter [1-12] - Simulation Result (Serpentine Scan)

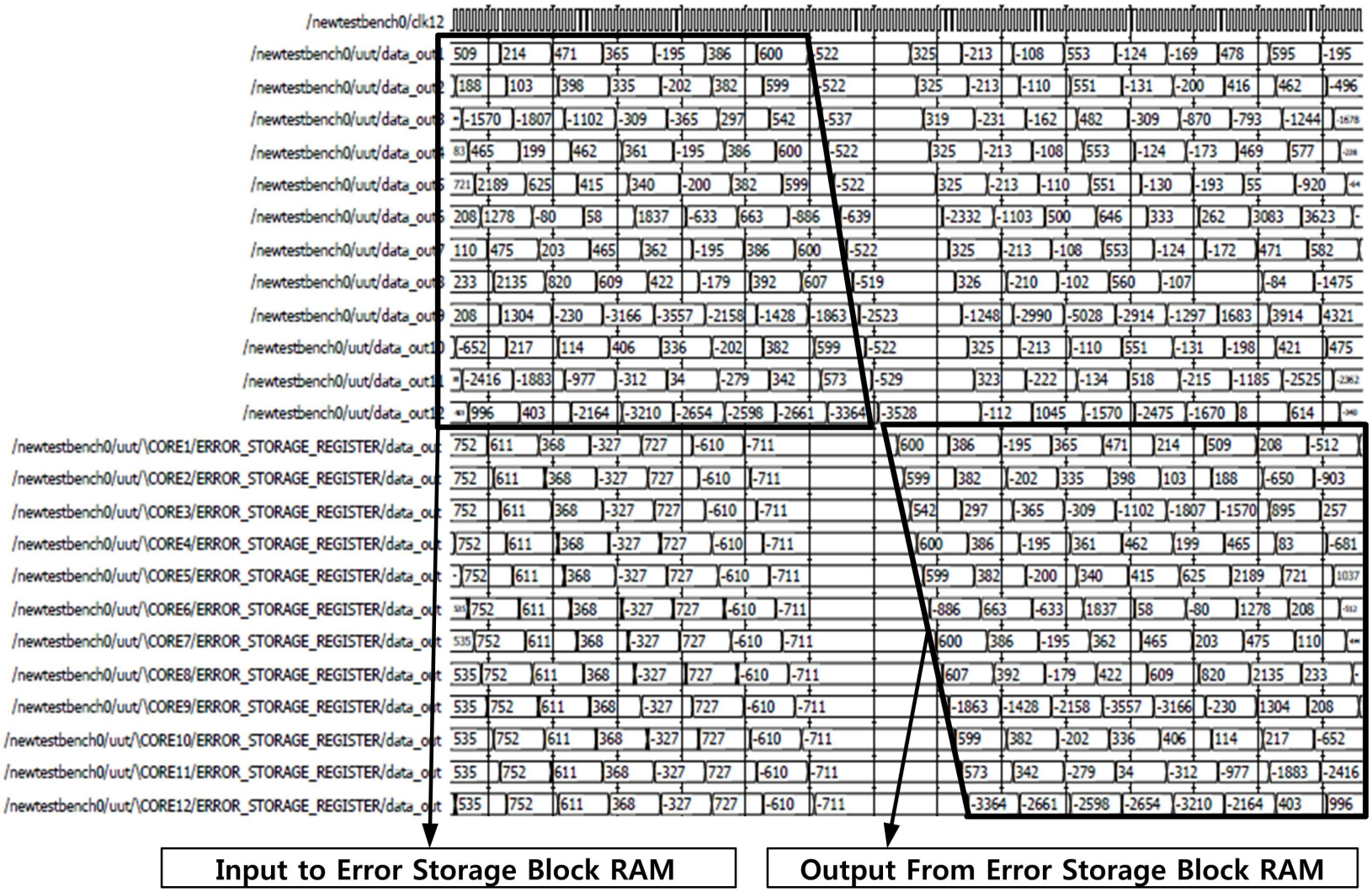

Figure 9.25: Error Storage Block RAM Data Buffering [1-12] - Simulation Result (Serpentine Scan)

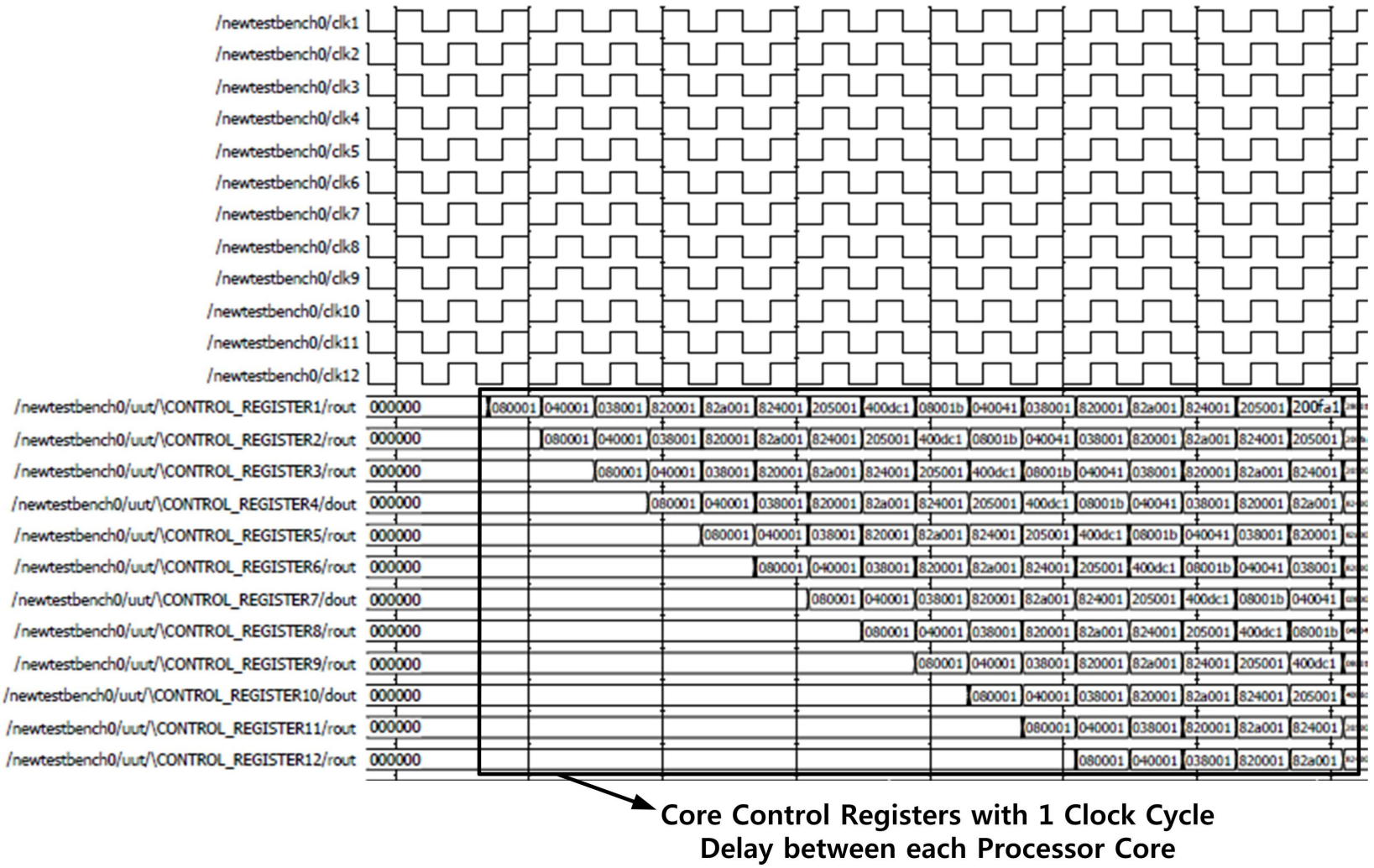

Figure 9.26: Processor Core Control Registers [1-12] - Simulation Result

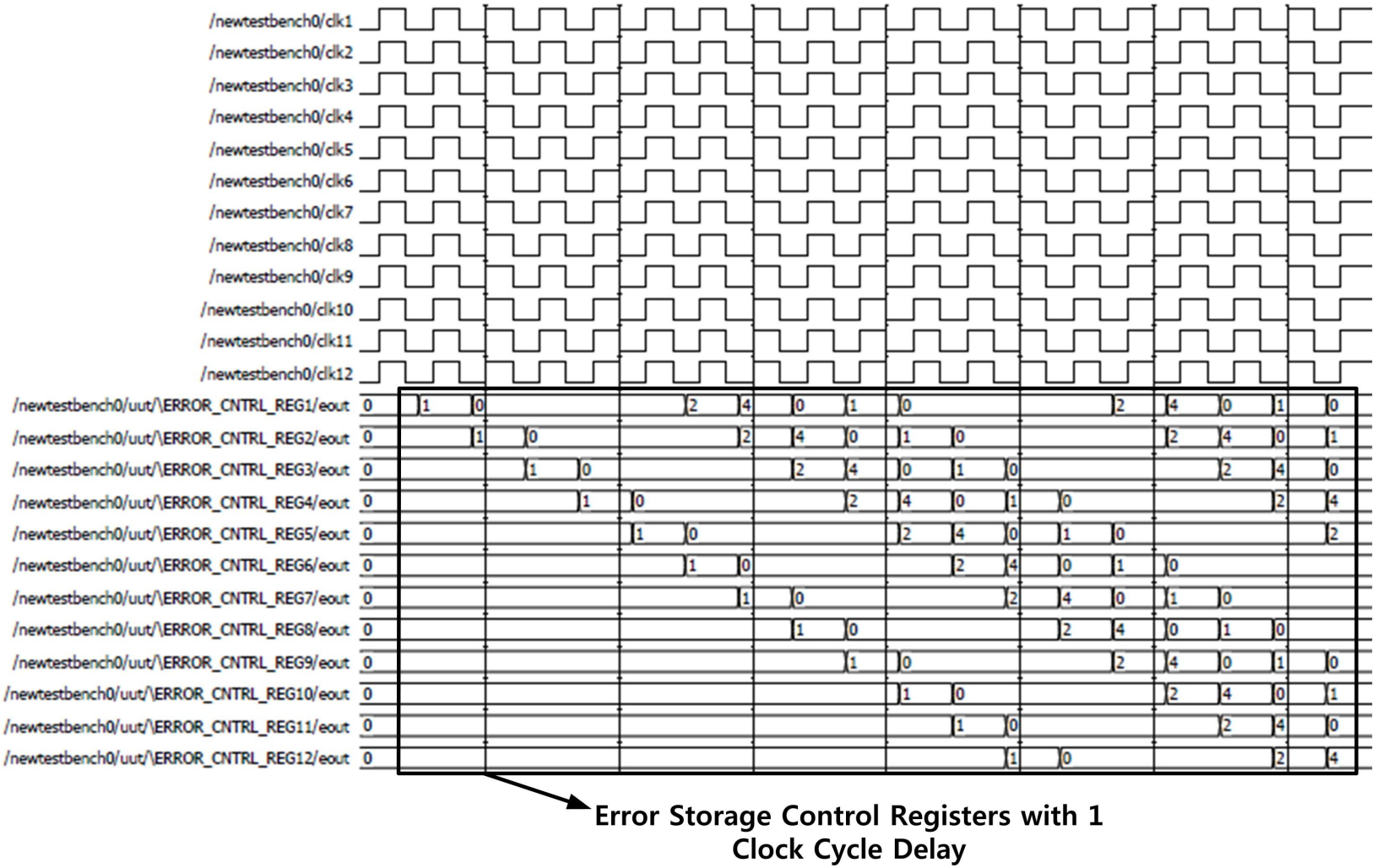

Figure 9.27: Error Storage Block Control Registers [1-12] - Simulation Result

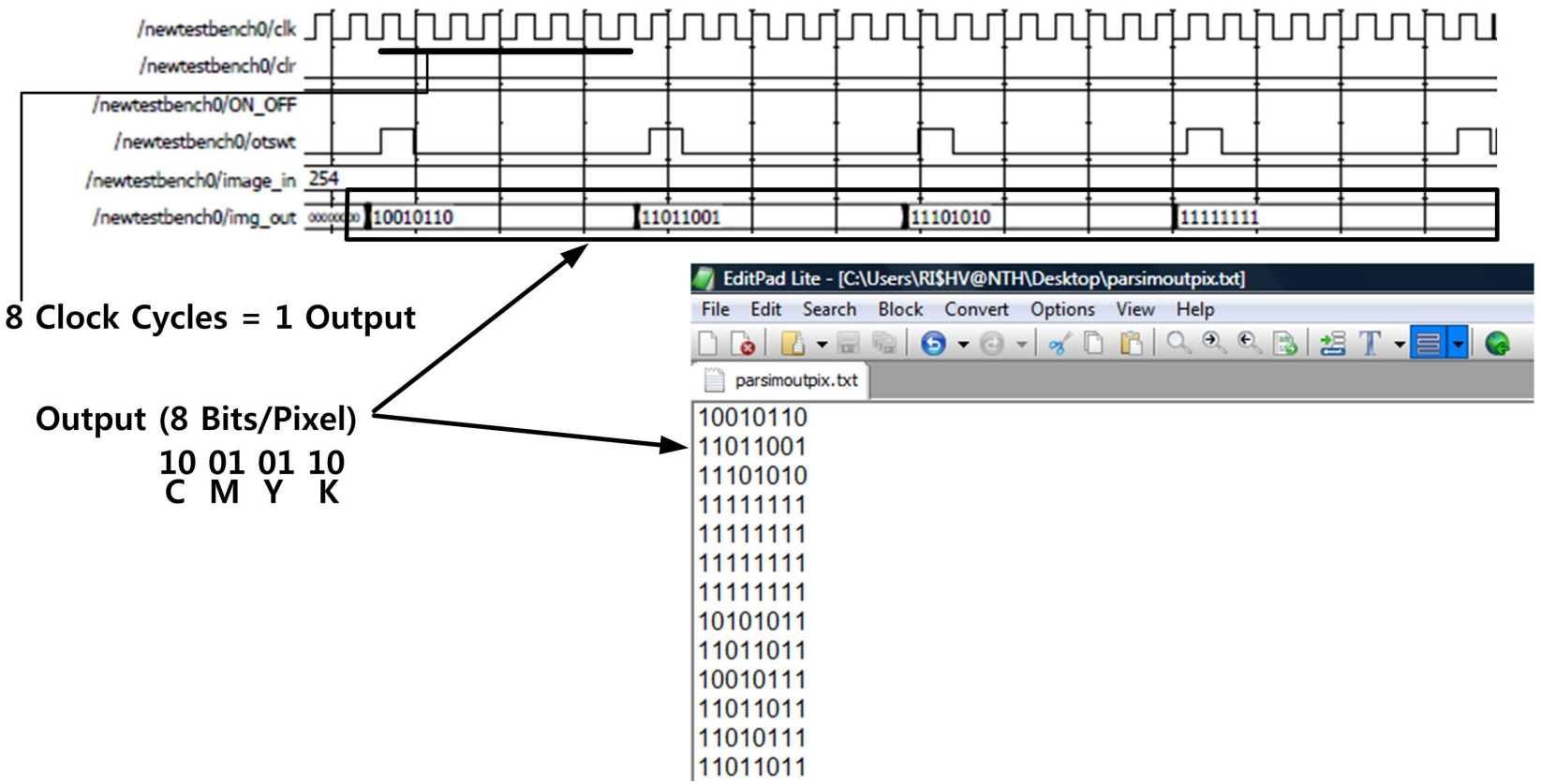

Figure 9.28: Halftoned Output Pixels - Simulation Results

9.4 **Output Images from Simulation Results**

Figure 9.29: Original Image (CMYK)

Figure 9.30: Halftoned Image (Software 'C' Code)

Figure 9.31: Halftoned Image (Hardware - FPGA)

Figure 9.32: Original Image (CMYK)

Figure 9.33: Halftoned Image (Software 'C' Code)

Figure 9.34: Halftoned Image (Hardware - FPGA)

Figure 9.35: Original Image (CMYK)

Figure 9.36: Halftoned Image (Software 'C' Code)

Figure 9.37: Halftoned Image (Hardware - FPGA)

Figure 9.38: Original Image

(GrayScale)

Figure 9.39: Halftoned Image

(Hardware - FPGA)

9.5 **Image Quality Comparison**

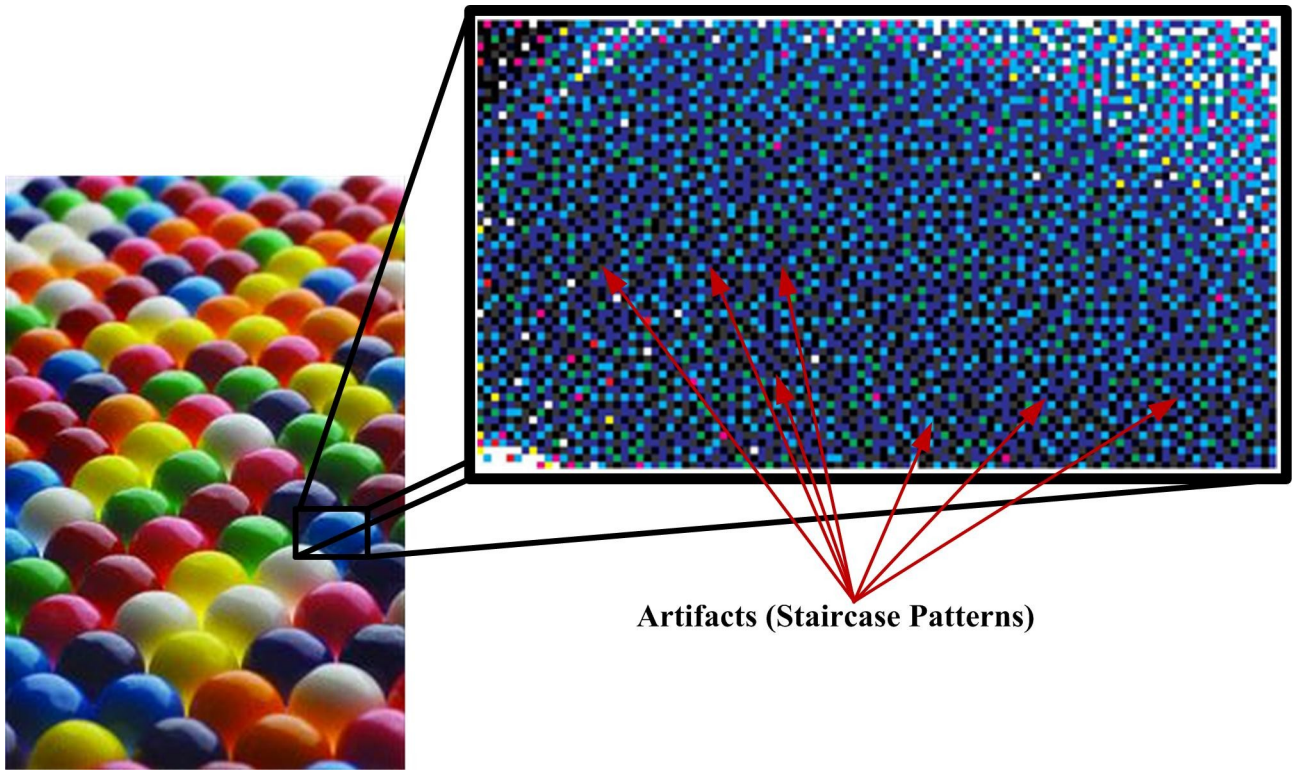

Figure 9.40: Halftoned Image by Binary Thresholding Technique - Zoomed Pixels Showing Artifacts

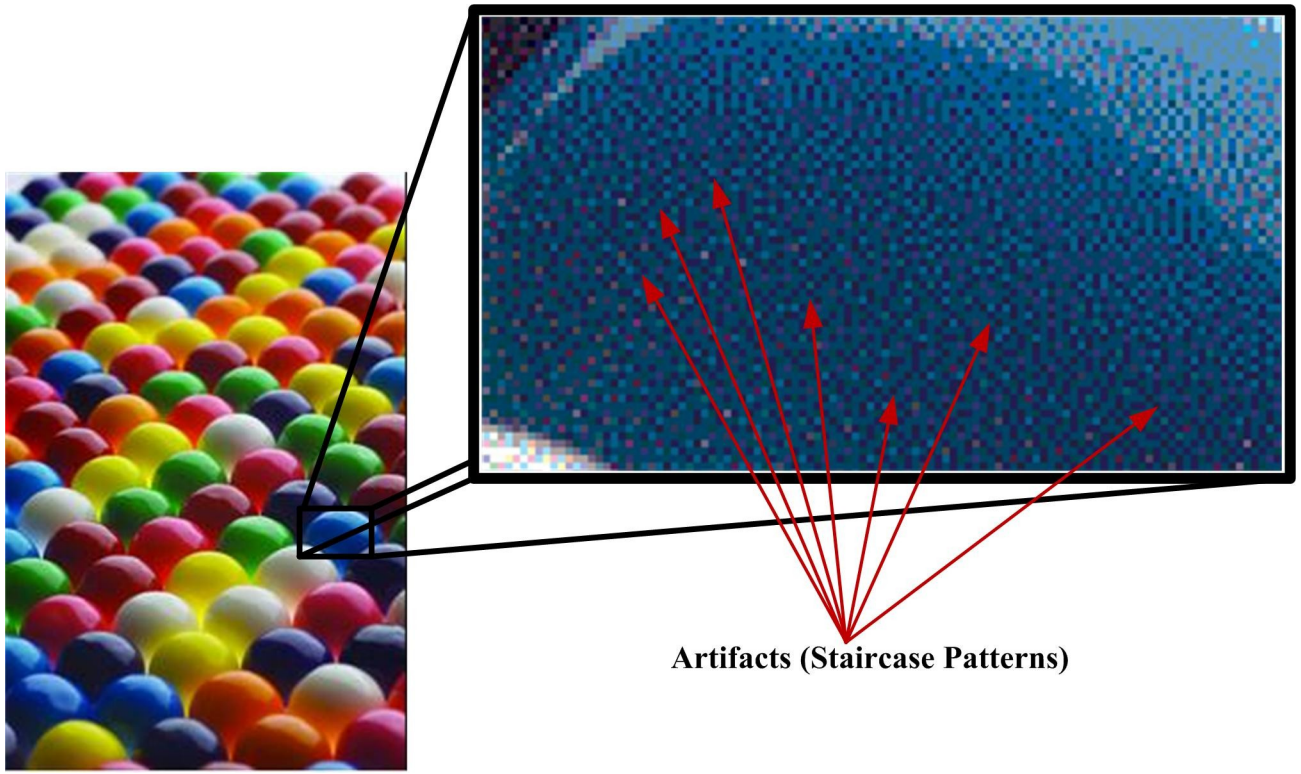

Figure 9.41: Halftoned Image by N-Level Quantization Technique - Zoomed Pixels Showing Artifacts

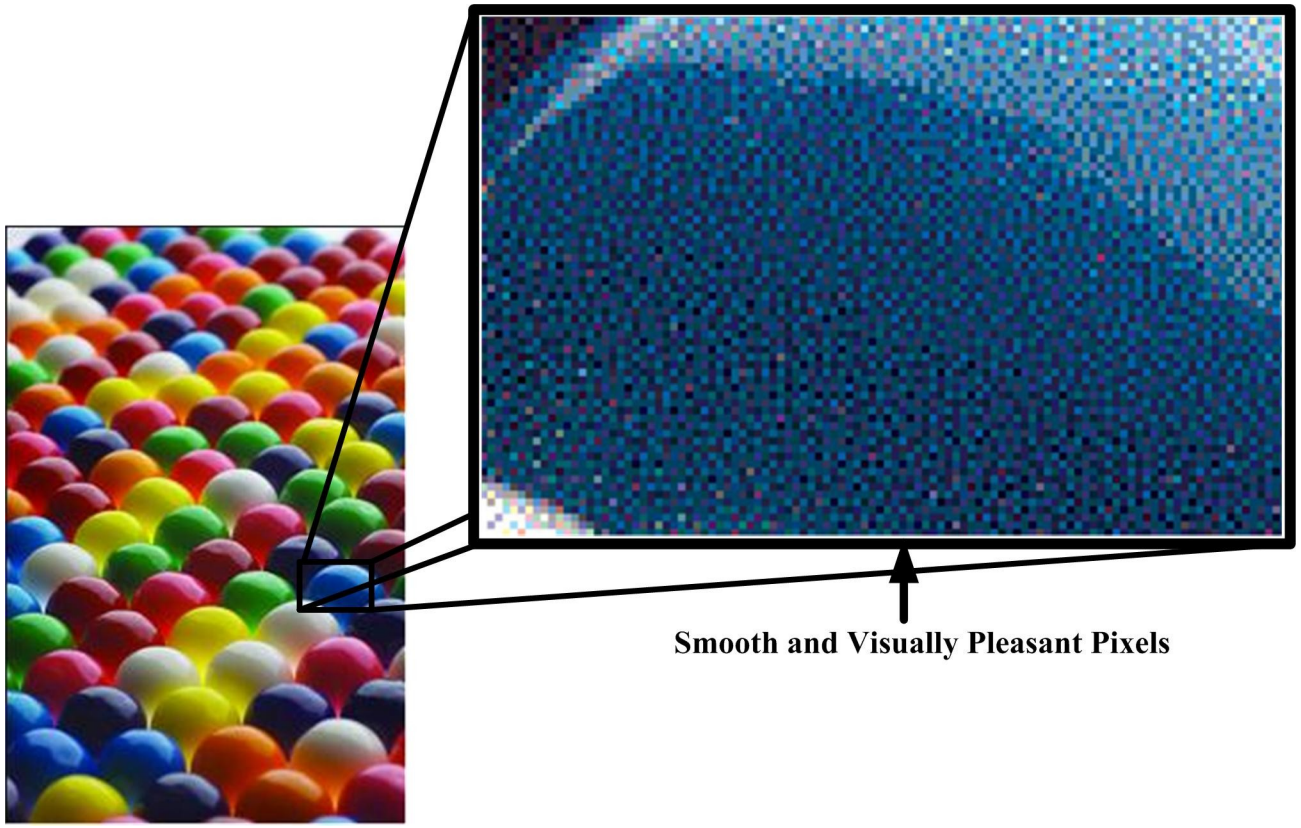

Figure 9.42: Halftoned Image by Stacked Error-Diffusion Technique (Software - 'C' Code - CPU) - Zoomed Pixels Showing Visually Pleasant Pixels

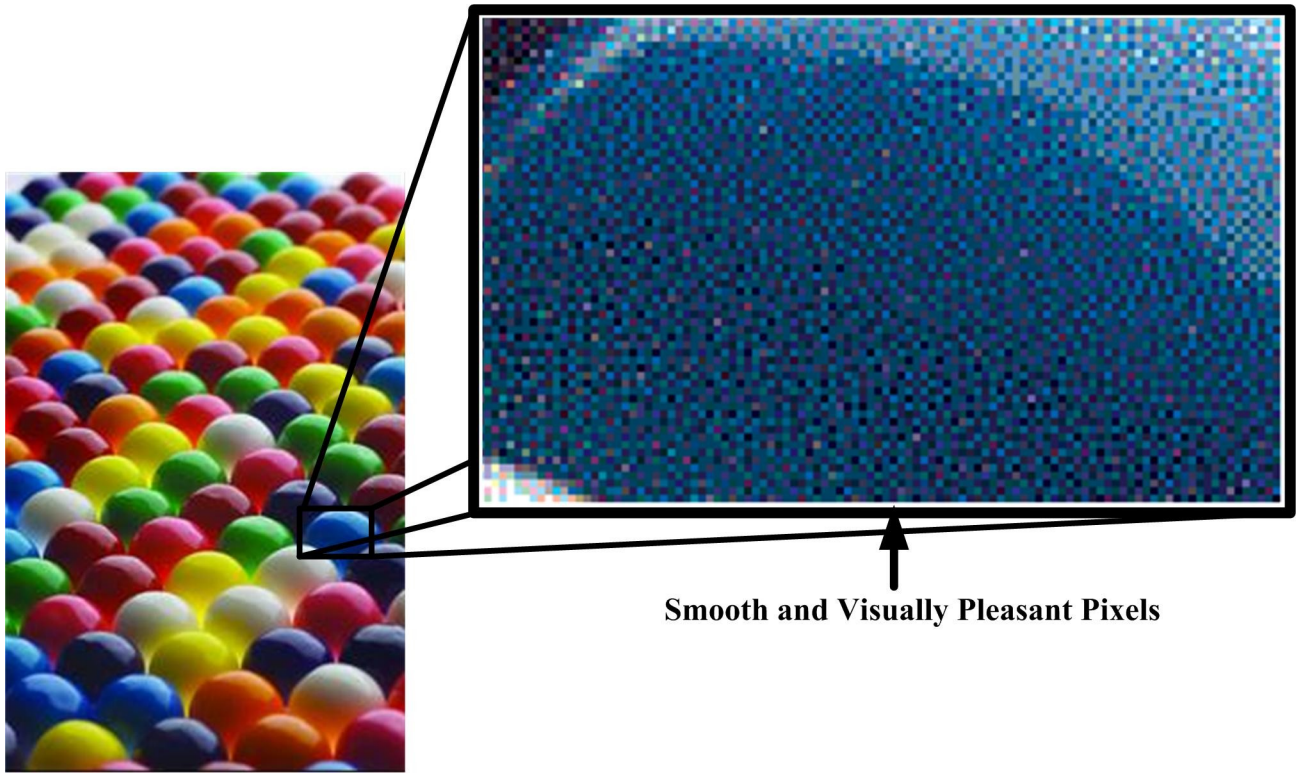

Figure 9.43: Halftoned Image by Stacked Error-Diffusion Technique (Hardware - FPGA) - Zoomed Pixels Showing Visually Pleasant Pixels

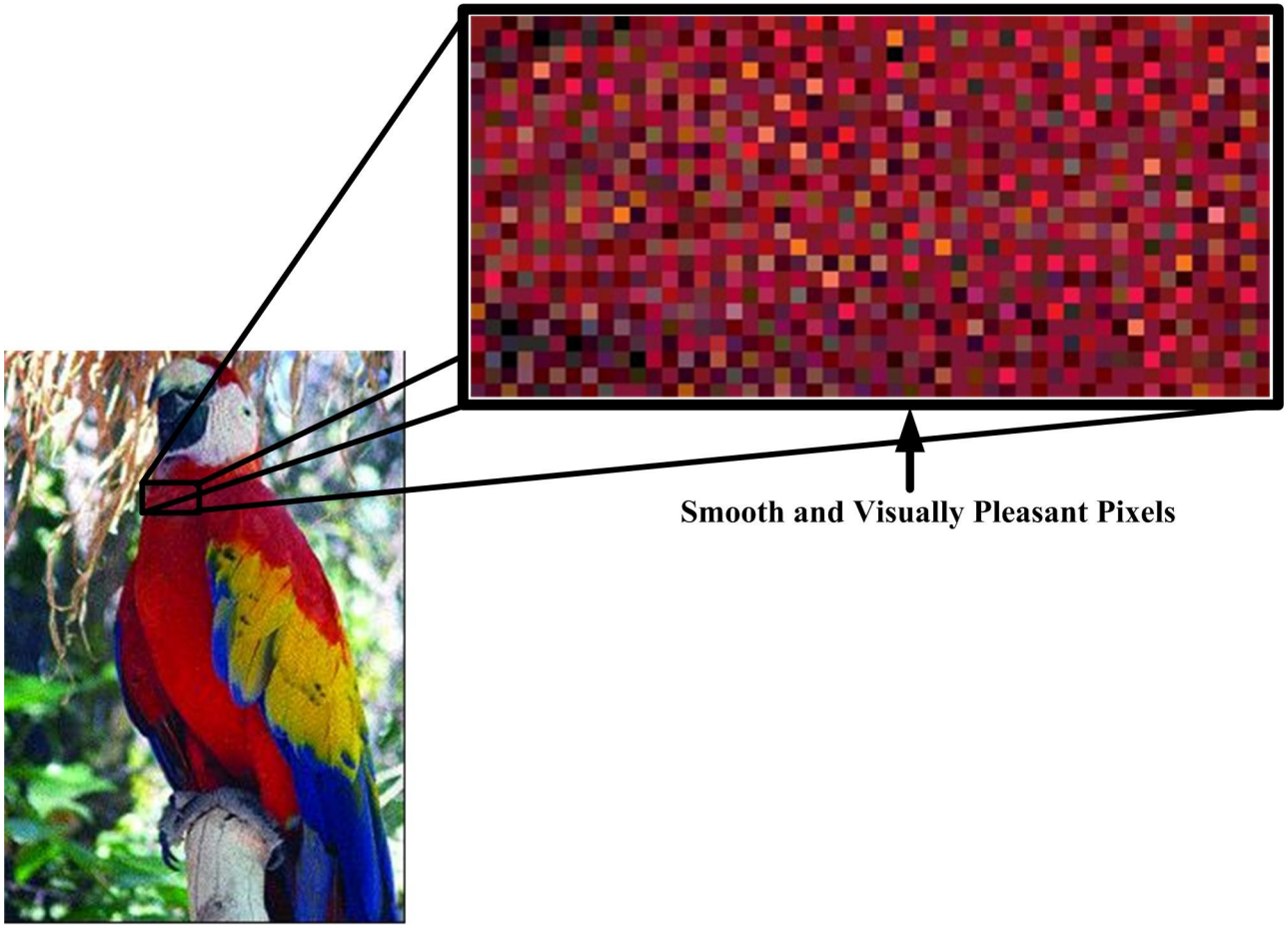

Figure 9.44: Halftoned Image by Stacked Error-Diffusion Technique (Software -'C' Code - CPU) - Zoomed Pixels Showing Visually Pleasant Pixels

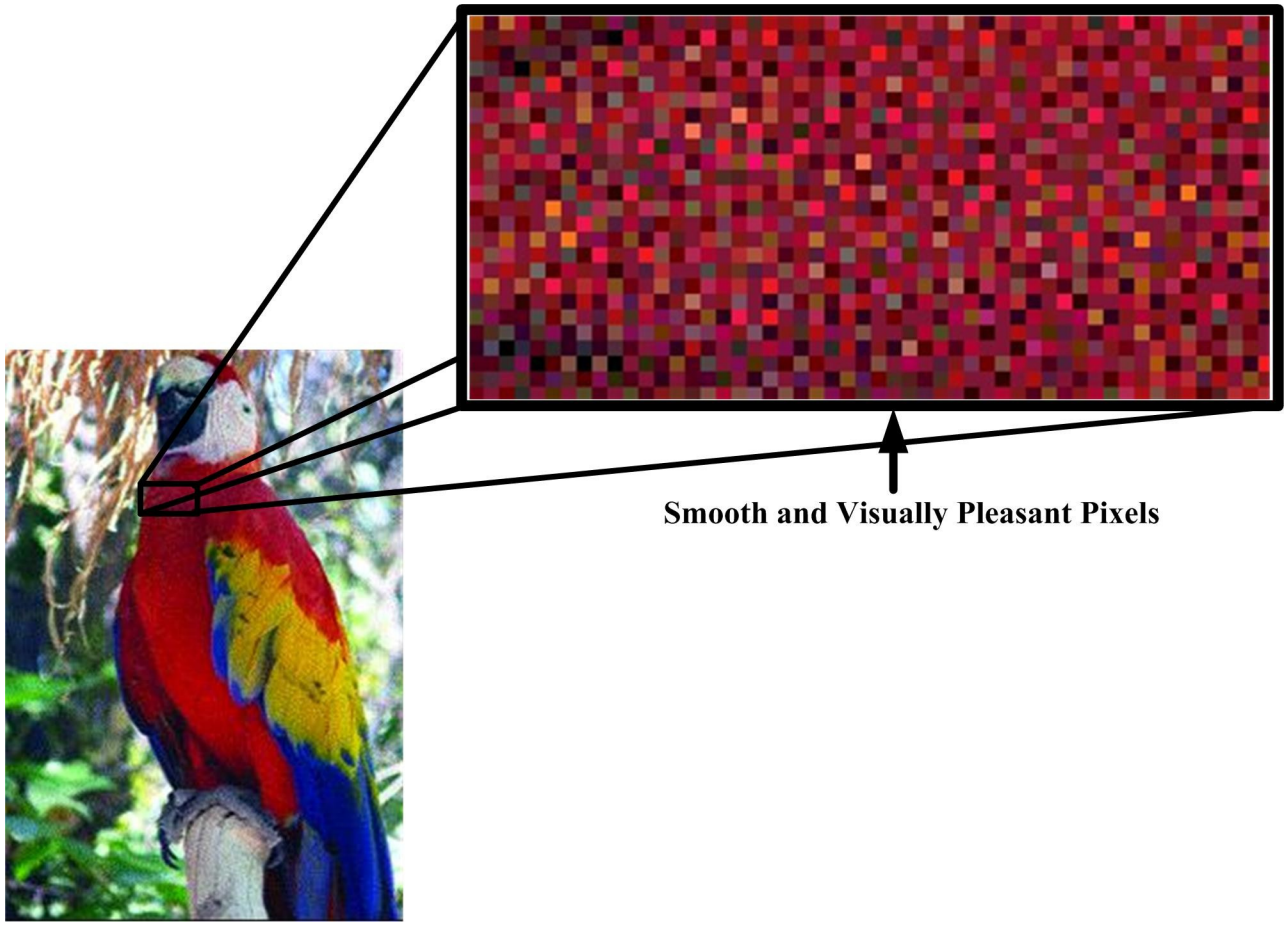

Figure 9.45: Halftoned Image by Stacked Error-Diffusion Technique (Hardware - FPGA) - Zoomed Pixels Showing Visually Pleasant Pixels

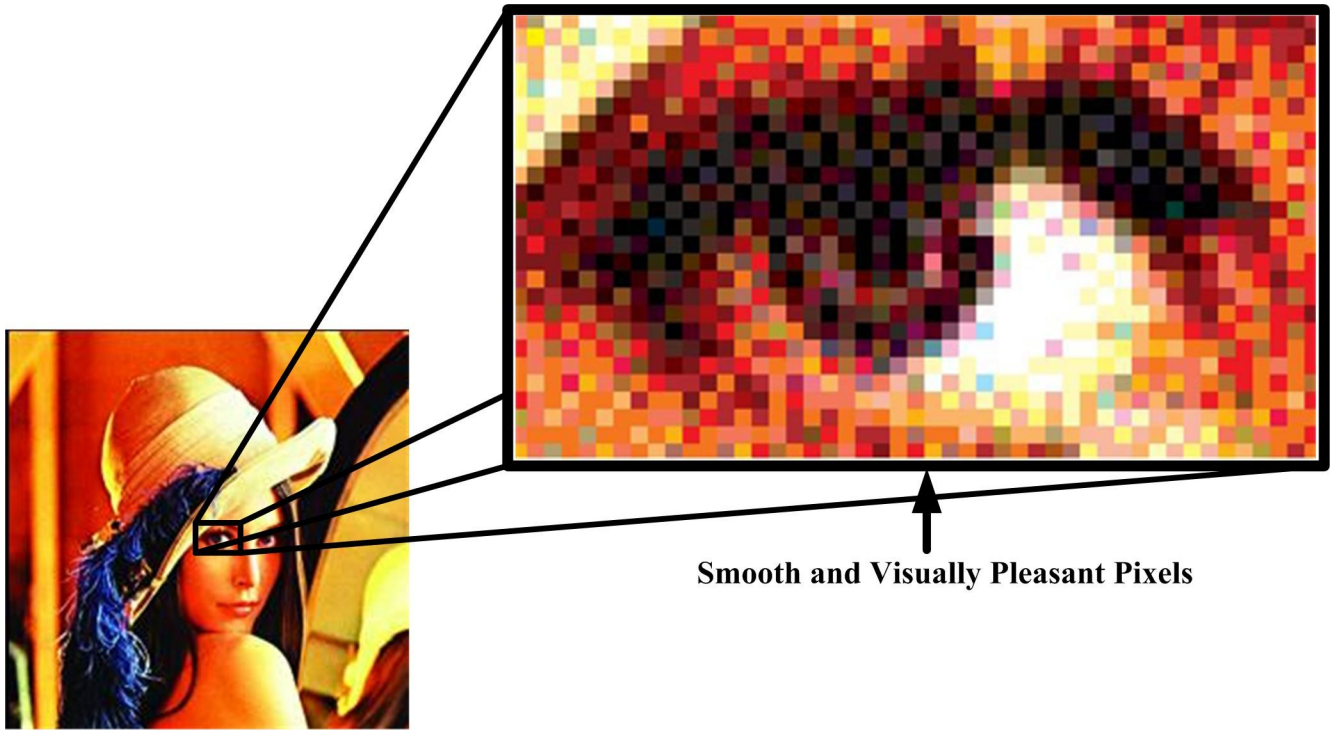

Figure 9.46: Halftoned Image by Stacked Error-Diffusion Technique (Software - 'C' Code - CPU) - Zoomed Pixels Showing Visually Pleasant Pixels

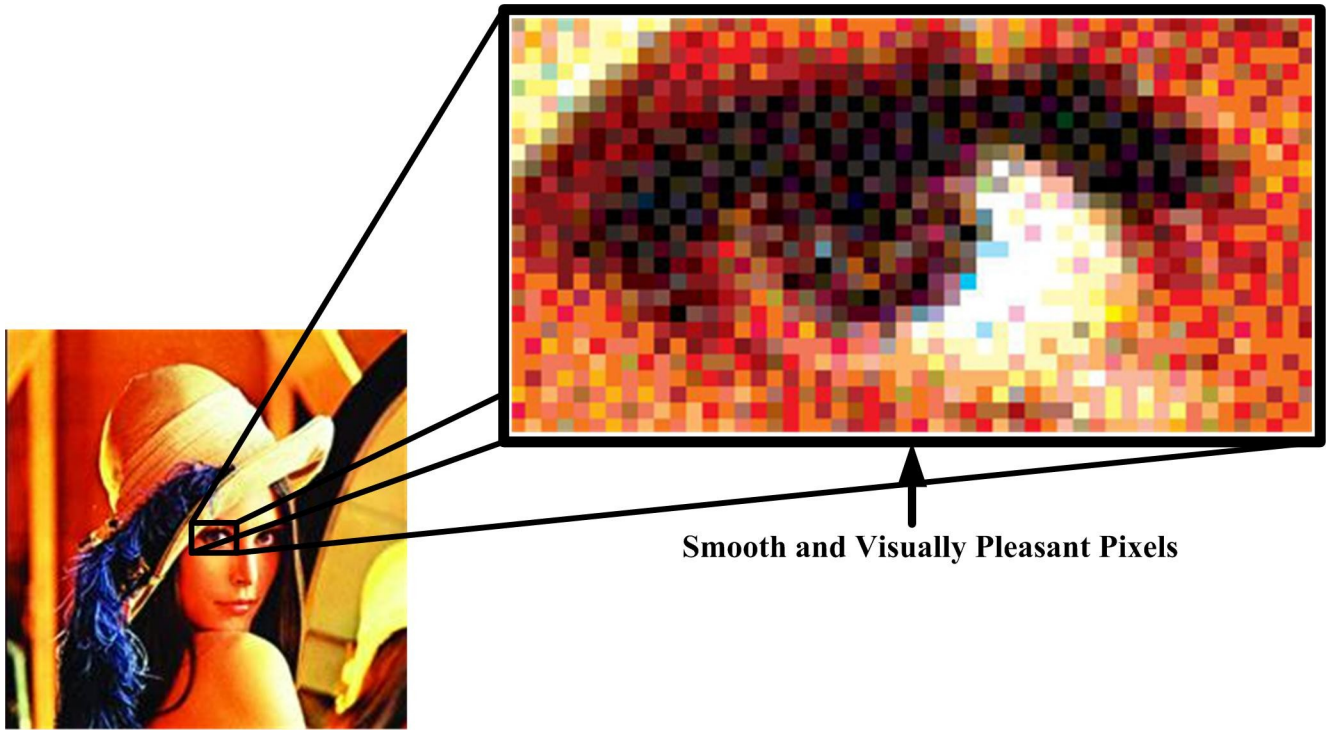

Figure 9.47: Halftoned Image by Stacked Error-Diffusion Technique (Hardware - FPGA) - Zoomed Pixels Showing Visually Pleasant Pixels

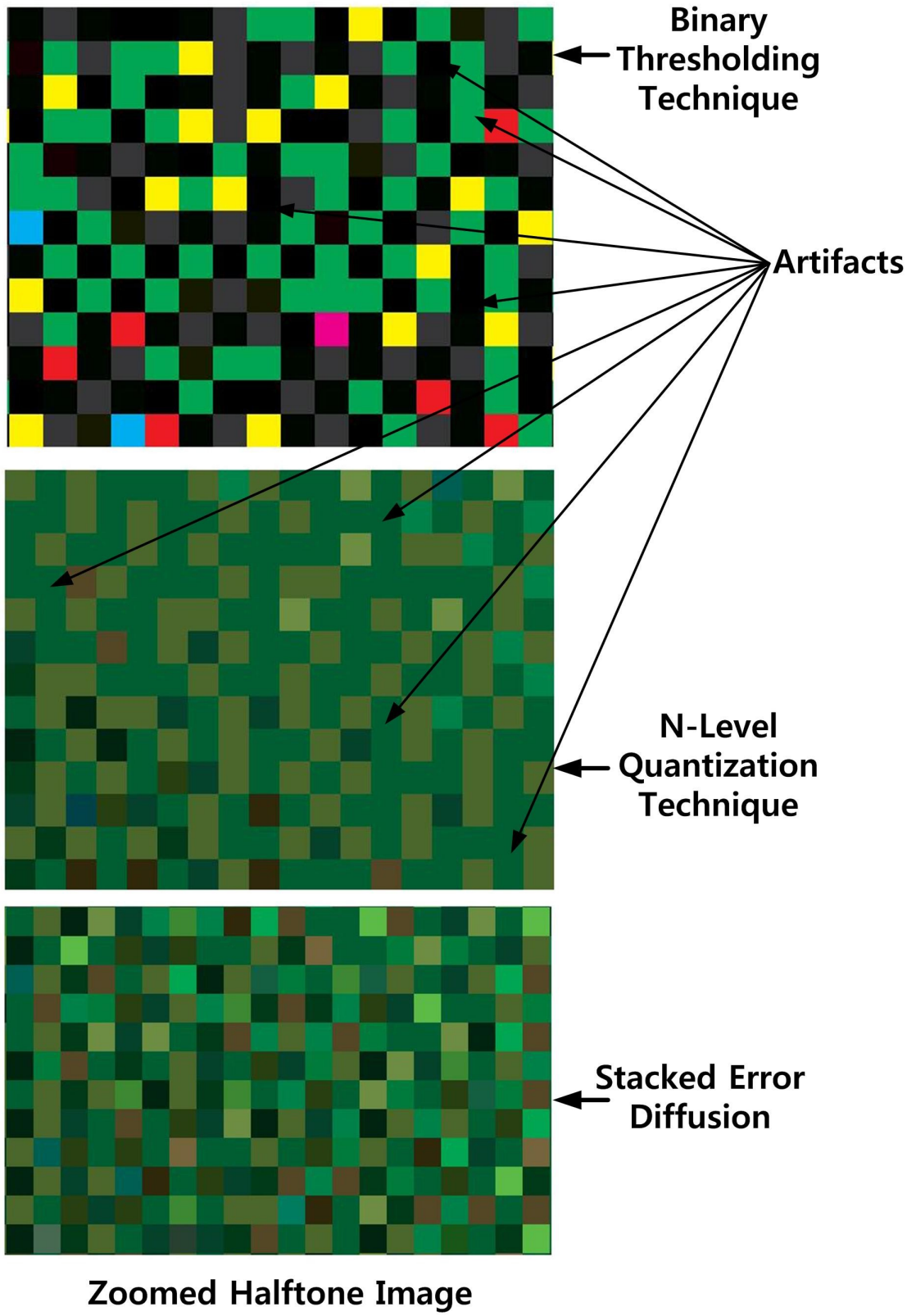

Figure 9.48: Zoomed Pixels showing Artifacts

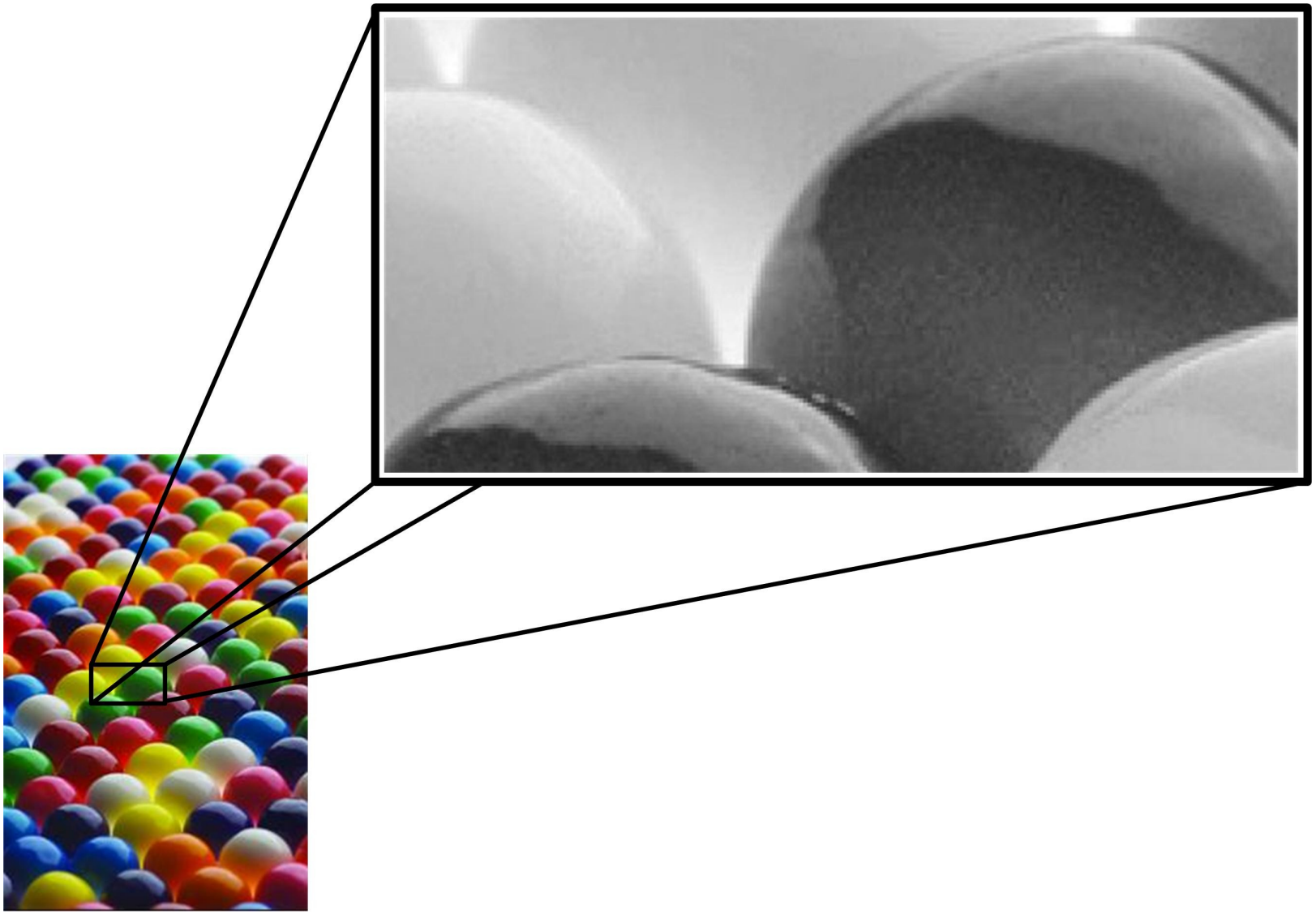

Figure 9.49: Zoomed Pixels of Original Image showing Cyan Color Only

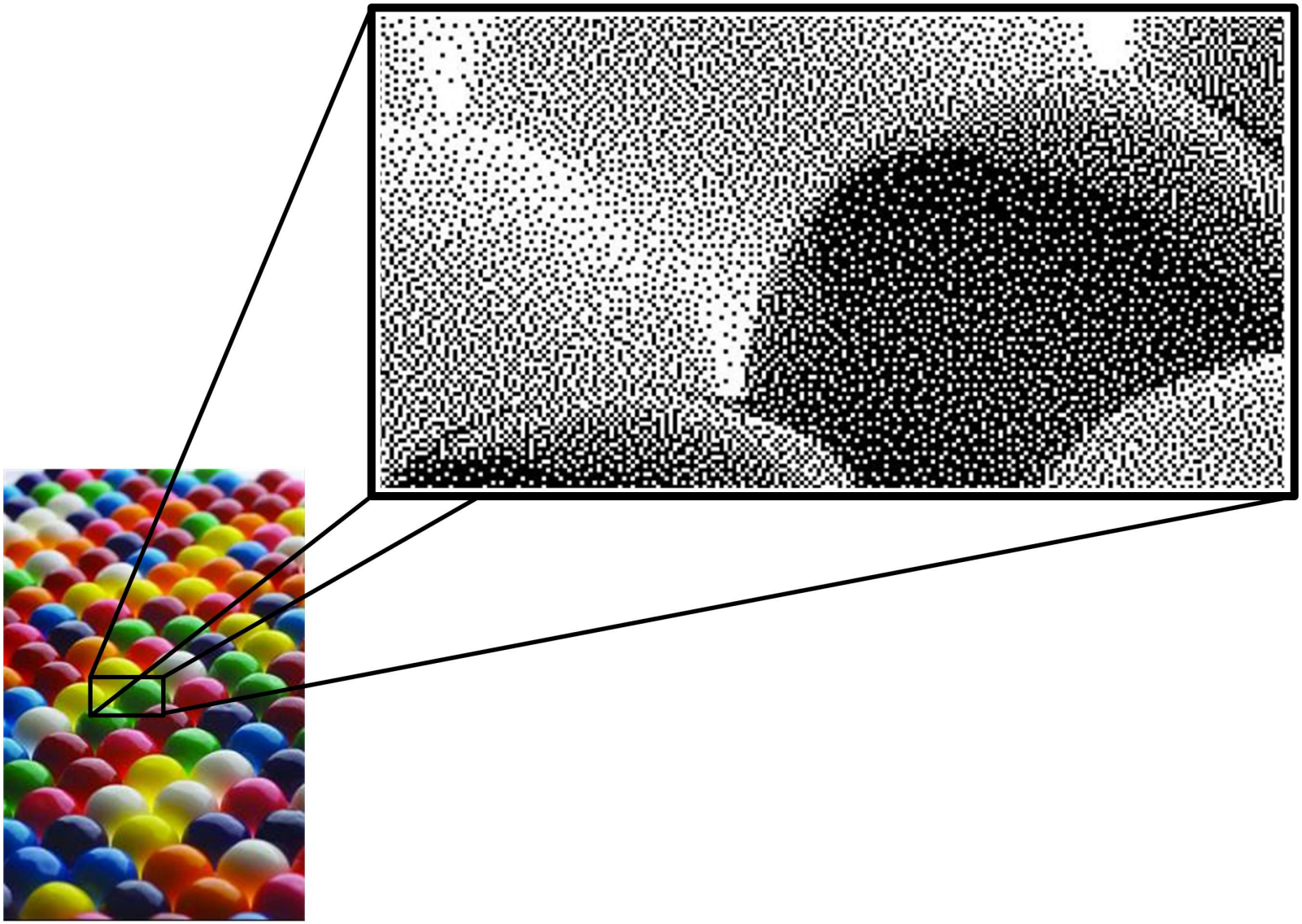

Figure 9.50: Zoomed Pixels of Halftoned Image Using Binary Thresholding Technique (Cyan Color Only)

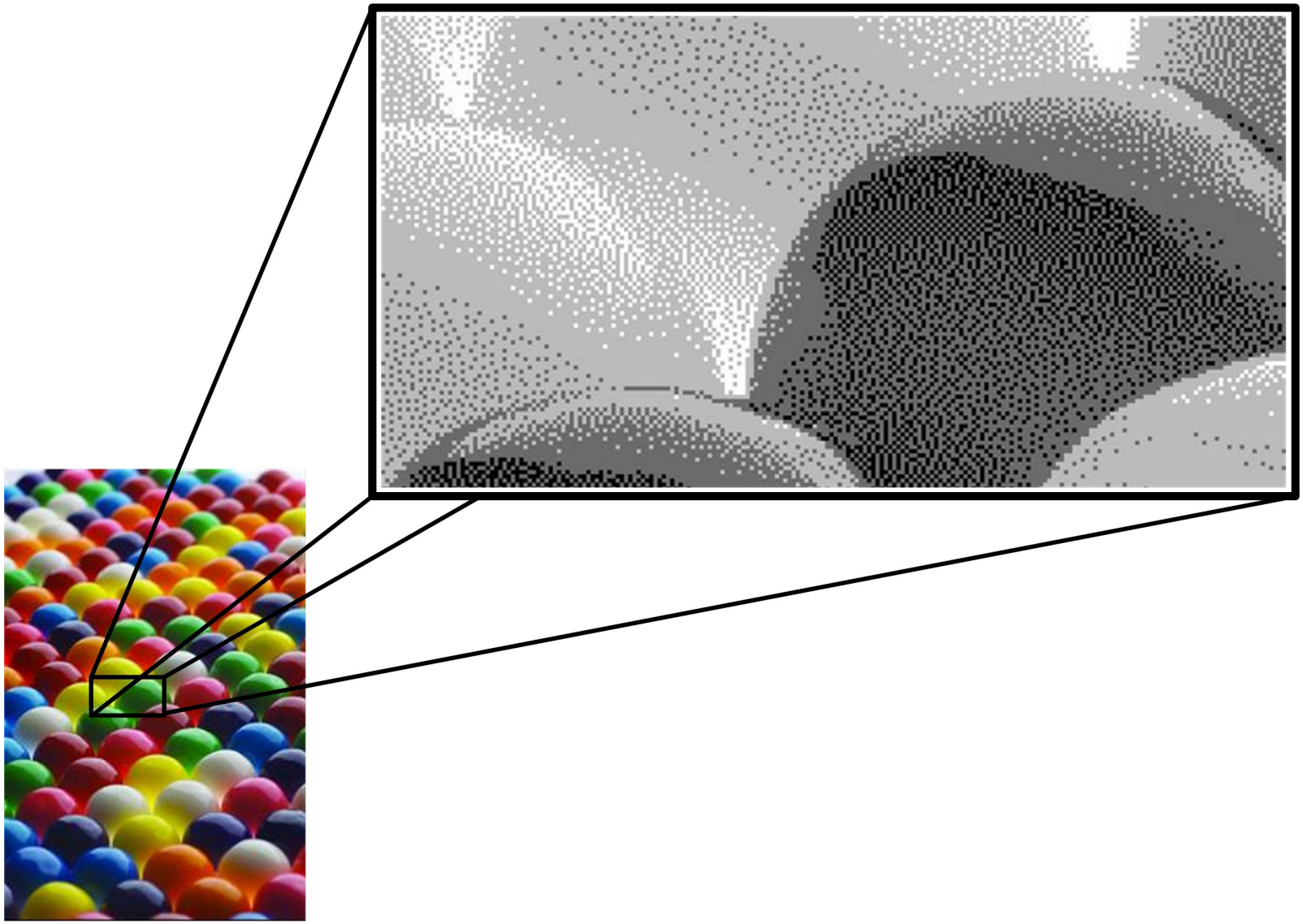

Figure 9.51: Zoomed Pixels of Halftoned Image Using N-Level Quantization Technique (Cyan Color Only)

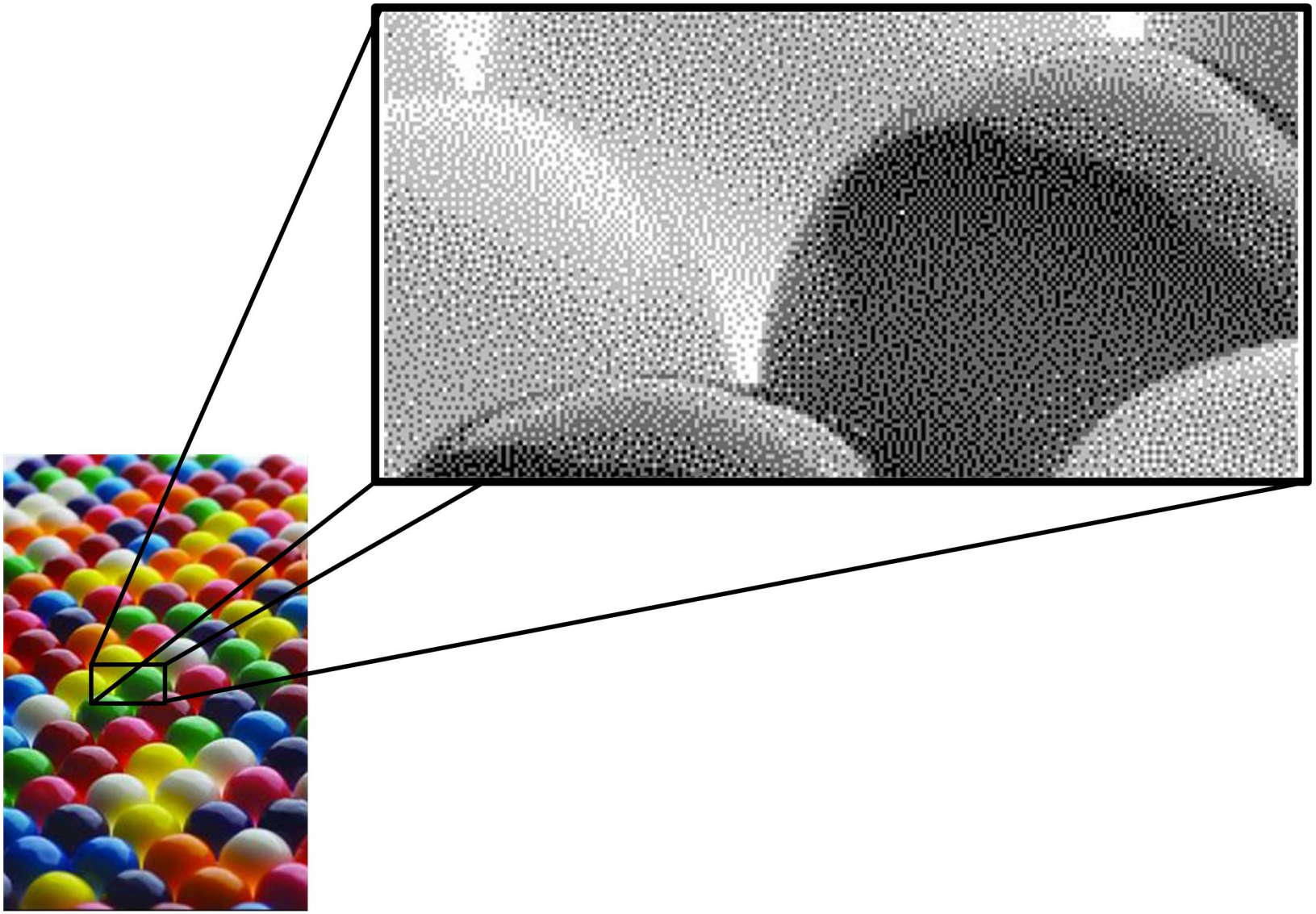

Figure 9.52: Zoomed Pixels of Halftoned Image Using Stacked Error-Diffusion 'C' Code (Cyan Color Only)

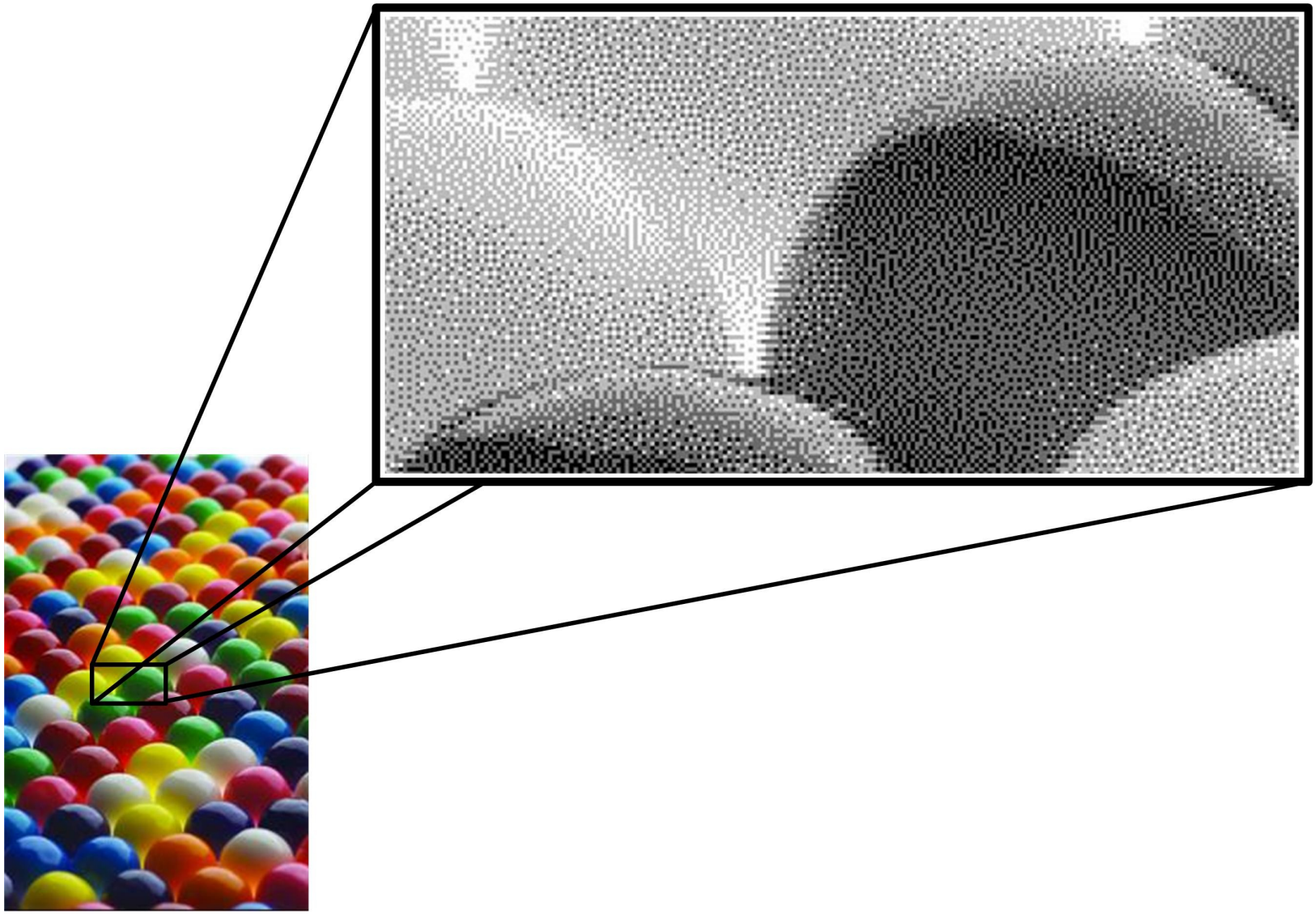

Figure 9.53: Zoomed Pixels of Halftoned Image Using Stacked Error-Diffusion Hardware-FPGA (Cyan Color Only)

The Figures 9.29 through 9.39 shows different images executed using Stacked Error-Diffusion Halftoning algorithm in a serial based CPU and in Parallel Hardware Architecture implemented in FPGA. The aim was to compare the halftoned output of an image executed in a serial based CPU with the halftoned output of the same image obtained from the Hardware Architecture in FPGA. Figure 9.29 shows the original color image (CMYK). Figure 9.30 shows the halftoned output of the image using Software 'C' code which was run in a serial based CPU. The pixels in the Figure 9.30 appears very smooth and pleasant to the naked eye. The same output was obtained when the original image was processed using FPGA as shown in Figure 9.31. The pixels in both the Figures 9.30 and 9.31 shows that the Halftoned image obtained is visually smooth and pleasant. Thus, the results obtained were the same from the FPGA HDL simulation when compared to the results obtained from the serial CPU without the loss in quality of the halftoned image. The comparison process was done using different images to ensure that the Hardware Architecture works for any input image. Figures 9.32 through 9.34 shows a different input image and the corresponding halftoned images. The halftoned output (Figure 9.34) when compared with the halftoned output in Figure 9.33 gives a clear idea of the accuracy obtained in the image quality. Figures 9.36 and 9.37 shows the halftoned outputs of the original image shown in Figure 9.35. Figure 9.38 shows a grayscale image that is used to show that the current Hardware Architecture supports any number of channels and levels. Figure 9.39 shows the halftoned output of the grayscale image that has only one channel and 3 levels per channel.

Figures 9.40 and 9.41 shows the halftoned output of the original image shown in Figure 9.32 using Binary Thresholding and N-Level Quantization halftoning algorithms. The Stacked Error-Diffusion algorithm is similar to Multitoning technique which results in an halftoned image of similar quality. A particular area of the in the image was marked and zoomed to show the difference in quality of the halftoned images obtained using Stacked Error-Diffusion Algorithm. The zoomed pixels in Figures 9.40 and 9.41 shows many artifacts (Staircase and Banding) present in Binary Thresholding and N-Level Quantization techniques. Figures 9.42 and 9.43 shows the zoomed pixels at the same

location using Stacked Error Diffusion technique executed in CPU and FPGA. It can be inferred that the pixels have the least number of artifacts (almost none) that results in smooth and visually pleasant halftoned image. Figures 9.44 through 9.47 shows pixels in the zoomed image and proves that the image obtained was smooth with fewer artifacts. Based on the halftoned images from the above figures mentioned, Figure 9.48 shows the zoomed version of all the algorithms compared with the Stacked Error-Diffusion Algorithm executed in hardware (Figure 9.34). From Figure 9.48, it can be concluded that artifacts (horizontal and vertical streaks of the same color intensity) are prevalent in all the other algorithms except the Stacked Error-Diffusion Algorithm (no two neighboring color intensities are the same, it appears visually smooth). Figure 9.49 shows the original continuous tone image (Cyan channel only) and the halftoned outputs of this image using different techniques in Figures 9.50, 9.51, 9.52 and 9.53. These figures show the comparison of different algorithms with the Stacked Error-Diffusion Algorithm taking only the Cyan channel. It can be inferred that Binary thresholding and N-Level Quantization technique results in banding artifacts [38] that degrades the image quality. It can be observed from the figures mentioned above that the Stacked Error-Diffusion Algorithm has fewer artifacts resulting in better image quality.

Chapter 10. Conclusions and Future Work

10.1 **Summary**

This thesis provided a detailed explanation of hardware and software techniques used to develop, simulate and validate a special purpose parallel architecture processor system that efficiently implements a new Stacked Error Diffusion Halftoning Algorithm. The hardware logic consumed by this architecture in the FPGA is described in detail in Appendix A. The introductory chapter at the beginning provided a thorough explanation of the Stacked Error-Diffusion algorithm. Chapter 2 gave a thorough insight into the data representation format used. Chapter 3 exclusively discussed the High Level System Architecture where the entire Hardware Halftoning System was shown (Figure 3.1). Chapters 4, 5, 6, 7 and 8 gave a comprehensive view of how the datapath and controller architecture was designed. Chapter 9 showed practical results obtained from the HDL simulations and compared and validated these results with the original results obtained from the algorithm executed on a serial commercially available CPU. The resulting parallel halftoning architecture this hardware design can be used to process wide images and print them with the help of the Wide Format Printers.

10.2 **Contributions**

- The Halftoning Algorithm written in 'C' was converted to an equivalent High Speed Hardware Parallel Architecture Design and Implemented into a Virtex-5 FPGA chip without compromising the Image Quality.
- A significant performance improvement can be achieved by increasing the execution speed of the algorithm by implementing if in the parallel architecture system implemented into a FPGA chip. Execution speed-up of 18 X is obtained by the algorithm implemented into a FPGA chip when compared to a conventional serial CPU.

- A very High Performance Parallel Hardware Architecture for implementation of new Stacked Error-Diffusion Halftoning Algorithm was designed, developed and validated.
- The entire system along with all the digital components required to develop the system were HDL simulated, tested and validated.
- The results obtained from the HDL simulation were compared and validated with the results from the original algorithm running on a serial general purpose CPU.

10.3 **Conclusion and Future Work**

This research accomplished the objectives addressed in Chapter 1. Output images obtained from the new hardware architecture was tested, evaluated and validated to be correct. The entire system runs at 50 MHz and can even run at a higher speed of 130 MHz. The base clock frequency of 50 MHz chosen for the new architecture produces the output twice as fast as a printer can print the data. This results in the avoidance of buffering problems between the printer and the FPGA. The software halftoning algorithm implemented in a commercial general purpose processor was very slow so that the printer had to stop for the pixels to be processed and then buffered into its memory. A substantial increase in throughput (12 square inches per second, 18X Speed) was achieved using the hardware implementation. The developed hardware unit was designed using the Xilinx ISE 10.3 CAD tool set [19] and simulated with the help of Mentor Graphics ModelSim CAD tool HDL simulator [20].

The next phase of this research work is to build a hardware prototype and test it connecting it to a Wide Format Printer. To extract the input image pixels from the host PC, an interface preferably PCI Express (because of higher speed) should be designed. A DDR2 SDRAM interface must be designed to continuously buffer the pixels from the host PC to the FPGA. SRAM (used to buffer pixels from the host PC) would be a better choice when compared to DRAM as it is faster and more efficient. The goal is for the entire architecture to run in a portable processing card known as PICO E-17 [37] which

has all the components required namely Virtex-5 FPGA, DDR2 SDRAM, PCIe, Ethernet and a Flash ROM to store the FPGA bit images. The speed of the architecture can be changed according to requirements. This research project met all original objectives. The hardware architecture was designed to be flexible and scalable. This architecture in addition to being implemented into FPGA technology can also be implemented to an Application Specific Integrated Circuit (ASIC) to achieve maximum performance.

Appendix A

Map Report

Release 10.1.03 Map K.39 (nt)

Xilinx Map Application Log File for Design 'Processor'

Design Information

Command Line:map -ise "C:/Documents and Settings/Rishvanth/Desktop/Error_diffusion_system/ERR_DIFF_SYSTEM.ise" -intstyle ise -p xc5vfx70t-ff665-1 -w -logic_opt off -ol high -t 1 -cm area -pr off -k 6 -lc off -power off -o Processor_map.ncd Processor.ngd Processor.pcf

Target Device : xc5vfx70t

Target Package : ff665

Target Speed : -1

Mapper Version : virtex5 -- $Revision: 1.46.12.2 $

Mapped Date : Thu Nov 04 18:14:35 2010

Mapping design into LUTs...

Running directed packing...

Running delay-based LUT packing...

INFO:Map:215 - The Interim Design Summary has been generated in the MAP Report (.mrp).

Running timing-driven packing...

Phase 1.1

Phase 1.1 (Checksum:2e20b97) REAL time: 42 secs

Phase 2.7

Phase 2.7 (Checksum:2e20b97) REAL time: 42 secs

Phase 3.31

Phase 3.31 (Checksum:2e20b97) REAL time: 42 secs

Phase 4.33

Phase 4.33 (Checksum:2e20b97) REAL time: 1 mins 5 secs

Phase 5.32

Phase 5.32 (Checksum:2e20b97) REAL time: 1 mins 7 secs

Phase 6.2

....

Phase 6.2 (Checksum:2fe3f91) REAL time: 1 mins 11 secs

.....

.....

.....
Phase 7.30
Phase 7.30 (Checksum:2fe3f91) REAL time: 3 mins 43 secs

Phase 8.3
...
Phase 8.3 (Checksum:3570f59) REAL time: 3 mins 43 secs

Phase 9.5
Phase 9.5 (Checksum:3570f59) REAL time: 3 mins 43 secs

Phase 10.8
.....
.....
.....
.....
.....
.....
Phase 10.8 (Checksum:3c4ec1f2) REAL time: 6 mins 24 secs

Phase 11.29
Phase 11.29 (Checksum:3c4ec1f2) REAL time: 6 mins 24 secs

Phase 12.5
Phase 12.5 (Checksum:3c4ec1f2) REAL time: 6 mins 24 secs

Phase 13.18
Phase 13.18 (Checksum:3d244906) REAL time: 6 mins 55 secs

Phase 14.5
Phase 14.5 (Checksum:3d244906) REAL time: 6 mins 55 secs

Phase 15.34
Phase 15.34 (Checksum:3d244906) REAL time: 6 mins 55 secs
REAL time consumed by placer: 6 mins 58 secs

CPU time consumed by placer: 6 mins 54 secs

Design Summary

Design Summary:

Number of errors: 0

Number of warnings: 3

Slice Logic Utilization:

Number of Slice Registers:	3,647 out of 44,800 8%
Number used as Flip Flops:	3,647
Number of Slice LUTs:	2,947 out of 44,800 6%
Number used as logic:	2,733 out of 44,800 6%
Number using O6 output only:	2,373
Number using O5 output only:	78
Number using O5 and O6:	282
Number used as Memory:	210 out of 13,120 1%
Number used as Dual Port RAM:	210
Number using O6 output only:	30
Number using O5 output only:	30
Number using O5 and O6:	150
Number used as exclusive route-thru:	4
Number of route-thrus:	94 out of 89,600 1%
Number using O6 output only:	82
Number using O5 output only:	12

Slice Logic Distribution:

Number of occupied Slices:	1,616 out of 11,200 14%
Number of LUT Flip Flop pairs used:	4,508
Number with an unused Flip Flop:	861 out of 4,508 19%
Number with an unused LUT:	1,561 out of 4,508 34%
Number of fully used LUT-FF pairs:	2,086 out of 4,508 46%
Number of unique control sets:	249
Number of slice register sites lost to control set restrictions:	489 out of 44,800 1%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element. The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.

IO Utilization:

Number of bonded IOBs:	62	out of 360	17%

Specific Feature Utilization:

Number of BlockRAM/FIFO:	109	out of 148	73%
Number using BlockRAM only:	109		
Total primitives used:			
Number of 18k BlockRAM used:	217		
Total Memory used (KB):	3,906	out of 5,328	73%
Number of BUFG/BUFGCTRLs:	13	out of 32	40%
Number used as BUFGs:	13		
Number of DSP48Es:	108	out of 128	84%

Peak Memory Usage: 599 MB

Total REAL time to MAP completion: 7 mins 33 secs

Total CPU time to MAP completion: 7 mins 26 secs

Mapping completed.

See MAP report file "Processor_map.mrp" for details.

Place and Route Report

Release 10.1.03 par K.39 (nt)

Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

OPTI960:: Thu Dec 02 14:59:14 2010

par -w -intstyle ise -ol std -t 1 Processor_map.ncd Processor.ncd Processor.pcf

Constraints file: Processor.pcf.

Loading device for application Rf_Device from file '5vfx70t.nph' in environment C:\Xilinx\10.1\ISE.

"Processor" is an NCD, version 3.2, device xc5vfx70t, package ff665, speed -1

Initializing temperature to 85.000 Celsius. (default - Range: 0.000 to 85.000 Celsius)

Initializing voltage to 0.950 Volts. (default - Range: 0.950 to 1.050 Volts)

Device speed data version: "PRODUCTION 1.64 2008-12-19".

Device Utilization Summary:

Number of BUFGs	13	out of 32	40%
Number of DSP48Es	108	out of 128	84%
Number of External IOBs	62	out of 360	17%
Number of LOCed IOBs	0	out of 62	0%
Number of RAMB18X2s	109	out of 148	73%
Number of Slice Registers	3647	out of 44800	8%

```
   Number used as Flip Flops            3647
   Number used as Latches               0
   Number used as LatchThrus            0
   Number of Slice LUTS                 2947 out of 44800   6%
   Number of Slice LUT-Flip Flop pairs  4508 out of 44800  10%
Overall effort level (-ol):   Standard
Router effort level (-rl):    Standard
Starting initial Timing Analysis.  REAL time: 28 secs
Finished initial Timing Analysis.  REAL time: 28 secs
Starting Router
Phase 1: 43997 unrouted;       REAL time: 31 secs
Phase 2: 28801 unrouted;       REAL time: 33 secs
Phase 3: 6167 unrouted;       REAL time: 41 secs
Phase 4: 6167 unrouted; (59559)      REAL time: 47 secs
Phase 5: 6169 unrouted; (0)      REAL time: 51 secs
Phase 6: 6169 unrouted; (0)      REAL time: 51 secs
Phase 7: 0 unrouted; (0)      REAL time: 1 mins 17 secs
Updating file: Processor.ncd with current fully routed design.
Phase 8: 0 unrouted; (0)      REAL time: 1 mins 20 secs
Phase 9: 0 unrouted; (0)      REAL time: 1 mins 20 secs
Phase 10: 0 unrouted; (0)      REAL time: 1 mins 33 secs
Total REAL time to Router completion: 1 mins 34 secs
Total CPU time to Router completion: 1 mins 33 secs
Partition Implementation Status
-------------------------------
  No Partitions were found in this design.
-------------------------------
Generating "PAR" statistics.
**************************
Generating Clock Report
**************************
```

Clock Net	Resource	Locked	Fanout	Net Skew (ns)	Max Delay(ns)
clk2_BUFGP	BUFGCTRL_X0Y12	No	164	0.277	2.018
clk9_BUFGP	BUFGCTRL_X0Y1	No	164	0.421	2.072
clk3_BUFGP	BUFGCTRL_X0Y10	No	164	0.352	2.074
clk10_BUFGP	BUFGCTRL_X0Y31	No	164	0.382	2.022
clk5_BUFGP	BUFGCTRL_X0Y21	No	163	0.519	2.061
clk7_BUFGP	BUFGCTRL_X0Y5	No	164	0.432	1.961
clk11_BUFGP	BUFGCTRL_X0Y20	No	164	0.501	2.072
clk_BUFGP	BUFGCTRL_X0Y8	No	156	0.335	1.917
clk4_BUFGP	BUFGCTRL_X0Y9	No	164	0.393	2.015
clk12_BUFGP	BUFGCTRL_X0Y13	No	166	0.528	2.074
clk6_BUFGP	BUFGCTRL_X0Y7	No	164	0.387	1.924
clk1_BUFGP	BUFGCTRL_X0Y23	No	165	0.454	2.099
clk8_BUFGP	BUFGCTRL_X0Y25	No	164	0.501	2.048

* Net Skew is the difference between the minimum and maximum routing only delays for the net. Note this is different from Clock Skew which is reported in TRCE timing report. Clock Skew is the difference between the minimum and maximum path delays which includes logic delays.

Timing Score: 0

Asterisk (*) preceding a constraint indicates it was not met.

This may be due to a setup or hold violation.

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
OFFSET = IN 4 ns BEFORE COMP "clk"	SETUP	0.313ns	3.687ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk11"	SETUP	0.638ns	3.362ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk12"	SETUP	0.836ns	3.164ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk7"	SETUP	1.035ns	2.965ns	0	0
TS_clk4 = PERIOD TIMEGRP "clk4"	SETUP	1.237ns	17.526ns	0	0
20 ns HIGH 50%	HOLD	0.302ns		0	0
OFFSET = IN 4 ns BEFORE COMP "clk3"	SETUP	1.328ns	2.672ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk2"	SETUP	1.550ns	2.450ns	0	0
TS_clk12 = PERIOD TIMEGRP "clk12"	SETUP	1.597ns	16.806ns	0	0
20 ns HIGH 50%	HOLD	0.238ns		0	0
OFFSET = IN 4 ns BEFORE COMP "clk9"	SETUP	1.624ns	2.376ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk8"	SETUP	1.755ns	2.245ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk10"	SETUP	2.069ns	1.931ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk1"	SETUP	2.219ns	1.781ns	0	0
TS_clk10 = PERIOD TIMEGRP "clk10"	SETUP	2.406ns	15.188ns	0	0
20 ns HIGH 50%	HOLD	0.299ns		0	0

OFFSET = IN 4 ns BEFORE COMP "clk5"	SETUP	2.536ns	1.464ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk6"	SETUP	2.538ns	1.462ns	0	0
OFFSET = IN 4 ns BEFORE COMP "clk4"	SETUP	2.704ns	1.296ns	0	0
TS_clk = PERIOD TIMEGRP "clk"	SETUP	2.818ns	14.364ns	0	0
20 ns HIGH 50%	HOLD	0.452ns		0	0
TS_clk11 = PERIOD TIMEGRP "clk11"	SETUP	2.840ns	14.320ns	0	0
20 ns HIGH 50%	HOLD	0.348ns		0	0
TS_clk6 = PERIOD TIMEGRP "clk6"	SETUP	2.884ns	14.232ns	0	0
20 ns HIGH 50%	HOLD	0.336ns		0	0
TS_clk9 = PERIOD TIMEGRP "clk9"	SETUP	2.969ns	14.062ns	0	0
20 ns HIGH 50%	HOLD	0.311ns		0	0
TS_clk3 = PERIOD TIMEGRP "clk3"	SETUP	3.447ns	13.106ns	0	0
20 ns HIGH 50%	HOLD	0.234ns		0	0
TS_clk2 = PERIOD TIMEGRP "clk2"	SETUP	3.564ns	12.872ns	0	0
20 ns HIGH 50%	HOLD	0.229ns		0	0
TS_clk7 = PERIOD TIMEGRP "clk7"	SETUP	4.185ns	11.630ns	0	0
20 ns HIGH 50%	HOLD	0.311ns		0	0
TS_clk8 = PERIOD TIMEGRP "clk8"	SETUP	4.260ns	11.480ns	0	0
20 ns HIGH 50%	HOLD	0.323ns		0	0
TS_clk5 = PERIOD TIMEGRP "clk5"	SETUP	4.398ns	11.204ns	0	0
20 ns HIGH 50%	HOLD	0.315ns		0	0
TS_clk1 = PERIOD TIMEGRP "clk1"	SETUP	5.448ns	9.104ns	0	0

20 ns HIGH 50%	HOLD	0.328ns		0	0
OFFSET = OUT 260 ns AFTER COMP "clk12"	MAXDELAY	241.236ns	18.764ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk11"	MAXDELAY	247.665ns	12.335ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk8"	MAXDELAY	248.267ns	11.733ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk10"	MAXDELAY	248.526ns	11.474ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk9"	MAXDELAY	248.535ns	11.465ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk7"	MAXDELAY	248.942ns	11.058ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk3"	MAXDELAY	249.278ns	10.722ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk1"	MAXDELAY	249.423ns	10.577ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk4"	MAXDELAY	249.663ns	10.337ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk6"	MAXDELAY	249.852ns	10.148ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk5"	MAXDELAY	249.902ns	10.098ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk2"	MAXDELAY	249.957ns	10.043ns	0	0
OFFSET = OUT 260 ns AFTER COMP "clk"	MAXDELAY	250.549ns	9.451ns	0	0

All constraints were met.

Generating Pad Report.

All signals are completely routed.

Total REAL time to PAR completion: 1 mins 41 secs

Total CPU time to PAR completion: 1 mins 37 secs

Peak Memory Usage: 457 MB

Placer: Placement generated during map.

Routing: Completed - No errors found.

Timing: Completed - No errors found.

Number of error messages: 0

Number of warning messages: 0

Number of info messages: 0

Writing design to file Processor.ncd

PAR done!

References

[1] http://staffwww.itn.liu.se/~sasgo/TNM011/Digital_Halftoning

[2] R. W. Floyd and I. Steinberg, "An adaptive algorithm for spatial grayscale," *Proc. SID*, vol. 17, no. 2, pp. 75–78, 1976.

[3] R. A. Ulichney, "Dithering with blue noise," *Proc. IEEE*, vol. 77, no.1, pp. 56–79, Jan. 1988.

[4] J.B. Rodríguez, G.R. Arce and D.L. Lau, Blue-noise multitone dithering, *IEEE Trans. Image Process.* 17 (8) (2008), pp. 245–267.

[5] J. Sullivan, R. Miller, and G. Pios, "Image halftoning using a visual model in error difussion," *J. Opt. Soc. Amer. A*, vol. 10, no. 8, pp.1714–1724, Aug. 1993.

[6] R. Eschbach and K. T. Knox, "Error-diffusion algorithm with edge enhancement,"*J. Opt. Soc. Amer. A*, vol. 8, no. 8, pp. 1844–1850, Dec.1991.

[7] R. Eschbach, "Reduction of artifacts in error difussion by means of input-dependent weights," *J. Electron. Imag.*, vol. 2, no. 4, pp.352–358, Oct. 1993.

[8] V. Ostromoukhov, "A simple and efficient error-difussion algorithm," in *Proc. SIGGRAPH*, 2001, pp. 567–572.

[9] P. Li and J. P. Allebach, "Tone-dependent error difussion," *IEEE Trans. Image Process.*, vol. 13, no. 2, pp. 201–215, Feb. 2004.

[10] R. A. Ulichney, "Dithering with blue noise," *Proc. IEEE*, vol. 77, no.1, pp. 56–79, Jan. 1988.

[11] D. L. Lau and R. Ulichney, "Blue-noise halftoning for hexagonal grids," *IEEE Trans. Image Process.*, vol. 15, no. 5, pp. 1270–1284, May 2006.

[12] R. S. Gentile, E. Walowit, and J. P. Allebach, "Quantization and multilevel halftoning of color images for near-original image quality.," *J. Opt. Soc. Amer. A*, vol. 7, no. 6, pp. 1019–1026, Jun. 1990.

[13] F. Faheem, G. R. Arce, and D. L. Lau, "Digital multitoning using gray level separation," *J. Imag. Sci. Technol.*, vol. 46, no. 5, pp. 385–397, Sep./Oct. 2002.

[14] R. Miller and C. Smith, J. P. Allebach and B. E. Rogowitz, Eds., "Mean-preserving multilevel halftoning algorithm," in *Proc. SPIE: Human Vision. Visual Processing and Digital Display TV*, 1993, vol. 1913, pp. 367–377.

[15] P.T. Mataxas, "Parallel Digital halftoning by error diffusion", June 2003, ACM Proceedings of, the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge.

[16] Yuefeng Zhang, Line Diffusion: A Parallel Error Diffusion Algorithm for Digital Halftoning, The Visual Computer, 12 (1) 40-46, 1996.

[17] Jae-woo Ahn and Wonyong Sung, "Multimedia processor based implementation of an error-diffusion halftoning algorithm exploiting subword parallelism," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 16, no. 2, pp. 129-138, Feb. 2001.

[18] C.R. Brown and A. Savakis, "High-Performance Architecture for Color Error Diffusion," Proceedings of SPIE-IS&T Electronic Imaging, SPIE Vol. 5012, 2003.

[19] http://www.xilinx.com/support/download/index.htm

[20] http://model.com/content/modelsim-se-high-performance-simulation-and-debug

[21] http://www.xilinx.com/products/virtex5/index.htm

[22] D.A. Patterson and J.L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface," Morgan Kaufmann Publishers, 3rd ed., 2004.

[23] E.L. Oberstar, "Fixed-point representation & fractional math", Oberstar Consulting, 2007.

[24] Randy Yates, "Fixed-Point Arithmetic: An Introduction," *www.digitalsignallabs.com/fp.pdf* , 2009.

[25] Randy Yates, "Practical Considerations in Fixed-Point FIR Filter Implementations," *www.digitalsignallabs.com/fir.pdf* , 2009.

[26] http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[27] www.**xilinx**.com/itp/**xilinx**6/books/docs/cgn/cgn.pdf

[28] www.xilinx.com/support/documentation/user_guides/ug193.pdf

[29] V. Nelson, H. Nagle, B. Carroll, and J. Irwin, "Digital Logic Circuit Analysis and Design," Prentice Hall, 1995.

[30] *www.doe.carleton.ca/~jknight/97.478/97.478_03F/Advdig5cirJ.pdf*

[31] *http://www.ccse.kfupm.edu.sa/~elrabaa/coe202/Lessons/Lesson4_5.pdf*

[32] J.P. Hayes, "Computer Architecture and Organization," WCB / McGraw-Hill, 3rd ed., 1998.

[33] M.M. Mano, "Digital Logic and Computer Design," Prentice Hall, 2002.

[34] M.M. Mano, "Digital Design," Pearson Prentice Hall, 2007.

[35] http://www.asic-world.com/verilog/memory_fsm2.html

[36] D.A. Patterson and J.L. Hennessy, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers, 4th ed., 2007.

[37] http://www.picocomputing.com/support/E-17.php

[38] D.L. Lau and G.R. Arce, "Modern Digital Halftoning (Signal Processing and Communications)," CRC Press, 2nd ed., 2008.

VITA

Rishvanth Kora was born on May 31, 1985 in Sriharikota, Andhra Pradesh, India. The author received his Bachelor of Engineering (B.E.) degree in Electrical and Electronics from Anna University, Tamilnadu, India in the year 2006. He has worked as a software engineer in IBM, Bangalore, India before enrolling for Masters program in Electrical Engineering at University of Kentucky, Lexington. He has been working at Computer Architecture Laboratory as a Graduate Research Student under the guidance of Dr. J. Robert Heath since January 2009.