2002

Computing moments of a binary horizontally/vertically convex image using run-time reconfiguration

Cheowway Neoh
Louisiana State University and Agricultural and Mechanical College

COMPUTING MOMENTS OF A BINARY HORIZONTALLY/VERTICALLY CONVEX IMAGE USING RUN-TIME RECONFIGURATION

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Cheowway Neoh
B.S., Louisiana State University, 2000
May 2002

Acknowledgments

First and foremost, I would like to express my sincere appreciation and thanks to my advisor and major professor, Dr. Jerry Trahan, for his constant guidance and valuable comments, without which, this thesis would not have been successfully completed. My gratitude also goes out to Dr. Ramachandran Vaidyanathan and Dr. Suresh Rai for serving on my defense committee. I also wish to thank the members of my family, especially my mom and dad for always being there for me and act as my source of encouragement and inspiration. Last but certainly not least, I would also like to thank my group of friends here at LSU, whose company and friendship I dearly cherish. Thank you for making my stay here at LSU a memorable one.

Table of Contents

List of Tables

List of Figures

Abstract

In this thesis, we present a design for computing moments of a binary horizontally/vertically convex image on an FPGA chip, using run-time reconfiguration. We compute the moments of up to third order for a total of 16 moments. We address how run-time reconfiguration speeds up moment computations without taking up huge hardware resources. Since we are considering a binary horizontally/vertically convex image, we look at an alternative method in moment computations that utilizes constant coefficient multipliers. We divide the image into segments and process one segment at a time. We reconfigure the constant coefficient multipliers before processing the next segment. This thesis also looks at the interactions between different logic units for moment computations. We provide an estimate of the total number of CLBs used to implement this design on an FPGA chip. Finally, we address variations of this particular type of image, such as non-binary and non-convex and determine whether this design is still applicable in those instances.

Chapter 1

Introduction, Background, and Motivation

Image processing is an area that is becoming more popular due to the fact that visual information is the best form of information for human perception. Digital image processing, as the name implies, processes a digitized image and applies to various fields, such as medicine, engineering, geography, and archaeology. Application of digital image processing ranges from enhancing the quality of a picture to pattern recognition. Figure 1.1 shows the before and after enhancement of the picture "Lena" [WOLF]. After enhancement (right picture), we can distinguish the features clearly, as opposed to the blurry left picture.

Figure 1.1 Digital enhancement of a picture

Some applications depend on a vital property found in an image, known as moments [COC, CWH, DBM]. For example, moments extract certain features from a poor quality image. An algorithm can use these features to enhance the quality of the picture. In this thesis, we are trying to compute this property of an image. As with any

other problem, this has to be done at the least cost (in terms of time and hardware resources) possible, without any degradation in the performance of moment computations.

This thesis looks at computing moments of a specific type of image on a *Field Programmable Gate Array* (FPGA) chip. An FPGA is a type of programmable device. The main advantage of an FPGA is the ability to reconfigure certain hardware resources in the FPGA chip while executing other tasks, often called *run-time reconfiguration* (RTR) or "on the fly" reconfiguration. Functional density is a metric that measures the performance of computations using FPGAs [WH] and is defined as:

$$D = \frac{1}{A * T},$$

where A is the area of total hardware resources used to implement the computations and T is the total time taken to complete the computations. Since D is just the inverse of $A*T$, computations that have low area requirements and fast completion time will have a high functional density. Using RTR is an effective way of enhancing the functional density of FPGAs by allowing the implementation of many different logic functions with the same resources. Implementing a task on an FPGA chip reduces the need for a huge, static architecture since one can break the task into portions and reconfigure the hardware before processing the next portion.

To summarize, this thesis considers moment computations of a specific type of image on an FPGA chip, using run-time reconfiguration. We address how RTR can speed up moment computations. We also compare two systems used in computing moments and decide which system gives us the best area/time tradeoff. Then, we look closely at all the logic units in an FPGA chip and how they interact with each other to compute the moments.

1.1 Introduction to Programmable Logic Devices

Programmable devices are a class of general-purpose chips that can be configured for a wide variety of applications. The first programmable device that was introduced was the Programmable Read-Only Memory (PROM). PROMs contain programmable switches that are basically transistors that can be turned on and turned off by supplying a specific amount of current. PROMs, however, are less efficient in implementing logic circuits than later programmable devices. A Programmable Logic Device (PLD) consists of an array of programmable AND gates connected to an array of programmable OR gates. Programmable Array Logic (PAL) is a commonly used PLD consisting of a programmable AND-plane followed by a fixed OR-plane. PALs come in both mask and field versions. In the mask version, the manufacturer configures the chip, while field versions allow end users to program the chips. PAL is suitable for small logic circuits, while the Mask-Programmable Gate Array (MPGA) handles larger logic circuits. In 1985, Xilinx introduced FPGAs. Both MPGAs and FPGAs consist of logic blocks and interconnections among those blocks that are reprogrammable. The major difference between FPGAs and MPGAs is that an MPGA is programmed using integrated circuit fabrication to form metal interconnections while an FPGA is programmed via electrically programmable switches [BR, Hauck, VCC, VMS].

1.2 Overview of FPGAs

Four main categories of FPGAs are commercially available: symmetrical array, row-based, hierarchical PLD, and sea-of-gates, as shown in Figure 1.2 [VCC]. In all of these FPGAs, the interconnections and how they are programmed vary. Currently there

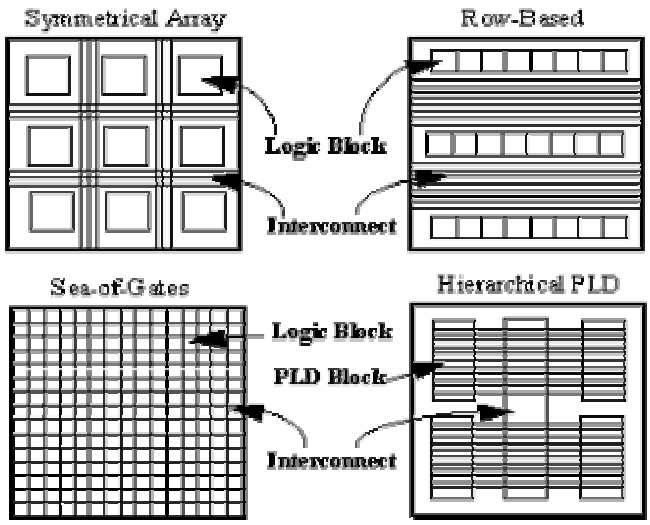

Figure 1.2 Classes of FPGA [VCC]

are three technologies in use: static RAM (SRAM) cells, anti-fuse, and Erasable Programmable Read-Only Memory (EPROM) / Electrically Erasable Programmable Read-Only Memory (EEPROM). Depending upon the application, one FPGA technology may have features desirable for that particular application. In SRAM technology, SRAM cells control the pass transistors, transmission gates, and multiplexers, which create the connections. A major advantage in using SRAM programming technology is that it allows fast reconfiguration, and the disadvantage of this technology is that it requires a large area. SRAM is also volatile and needs an external memory to program the chip on power up.

Anti-fuse is a technology that is not reprogrammable. It is basically a device with high resistance between its two terminals. When a high voltage is applied across the two terminals, a permanent link of low resistance will form between those two terminals. Hence, anti-fuse is a program-once device. EPROM and EEPROM devices use floating

gate technology. The advantages of both EPROM and EEPROM are that they are non-volatile and reprogrammable. EEPROM offers a slight advantage of being able to reconfigure electrically within the circuit instead of using UV light [BR, RGS, VCC]. Table 1.1 shows some of the commercially available FPGAs.

Table 1.1 Commercial FPGAs [VCC]

Company	Architecture	Logic Block Type	Programming Technology
Actel	Row-based	Multiplexer-Based	Anti-Fuse
Altera	Hierarchial-PLD	PLD Block	EPROM
QuickLogic	Symmetrical Array	Multiplexer-Based	Anti-Fuse
Xilinx	Symmetrical Array	Look-up Table	Static RAM

Figure 1.3 shows the basic architecture of a symmetrical array FPGA. This FPGA has four major components:

- configurable logic blocks (CLB),
- input-output (I/O) blocks,
- interconnect/routing resources, and
- RAM blocks.

Configurable logic blocks consist of look-up tables, multiplexers, flip-flops, and other logic units to implement different combinational and sequential logic functions. The I/O blocks provide an interface for incoming and outgoing signals for the FPGA. Routing resources provide the routing paths for inputs and outputs between two CLBs or a CLB and an I/O block. RAM memory blocks usually store intermediate results [RAM]. In our

thesis, we are looking specifically at the Virtex-E 1.8V FPGA by Xilinx [FPGA]. In Virtex-E, a CLB contains two slices, where each slice has identical logic units.

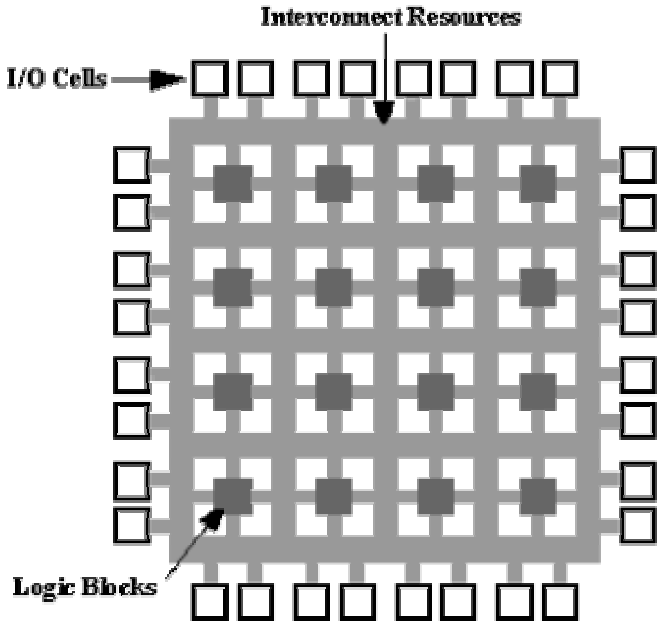

Figure 1.3 Basic FPGA architecture [VCC]

1.3 Introduction to Digital Image Processing

Digital image processing is a field that involves analyzing and processing images that have been converted to a numerical form. A two dimensional light intensity function, $f(i,j)$, represents most images, where i and j denote the spatial coordinates and the value of f at any point (i,j) is proportional to the brightness (or gray level) of the image at that point [GW]. Each element in the 2-D array is called a *pixel*. Typical size of an image are $N \times N$, where N = 256, 512 or 1024. The minimum value of a pixel is 0, which corresponds to a black color, while a pixel value of 255 represents a white color. The numbers between 0 and 255 describe the different shades of gray.

The two main purposes for digital image processing are to:

- Enhance the quality of images so that they can be viewed clearly without any strain on the eyes. Digital image processing extracts certain features from images and uses those features to produce the best images. The main aim of digital image processing is to enhance the pictorial information in images for human interpretation.
- Process acquired data for autonomous machine perception. Image processing extracts information from an image that is suitable for computer processing. Common tasks in machine perception that often utilize image processing techniques are automatic character recognition, industrial robots for product assembly and inspection, screening of x-rays and blood samples, and automatic processing of fingerprints.

The following table shows some areas that utilize digital image processing for specific applications [GW].

Table 1.2 Applications of digital image processing

Area	Application
Medicine	Enhancing x-rays and other medical images from MRI, scans, or ultrasounds
Architecture and Engineering	Enhancing architectural drawings and images of experiments in areas such as high-energy plasmas and electron microscopy
Geography	Studying pollution patterns from aerial and satellite images and predicting different types of weather phenomena
Archaeology	Restoring blurred pictures taken of rare artifacts

1.4 Overview of Moments

Moments are properties of an image. Numerous algorithms and techniques use moments for pattern recognition, object identification, 3-D object pose estimation, robot sensing, image coding, and reconstruction. Moments are very useful because their computation is algorithmically simple and uniquely defined for any image function. Besides that, moment-based methods often yield features from an image that are invariant to translation, scaling, and rotation [TC].

For a 2-D function $f(i,j)$ and an $N \times N$ image, a moment of order $\max\{p,q\}$ is defined as:

$$m_{pq} = \sum_{i=1}^{N} \sum_{j=1}^{N} i^p j^q f(i,j), \qquad \text{(Eq 1.1)}$$

for $1 \le i, j \le N$.

Sometimes, certain applications use binary images (images consisting of black and white pixels only) where a region R defines the shape of the object. All the pixels inside the region R will be black and pixels in the background (that is, outside R) will be white. In a binary image,

$$B(i,j) = \begin{cases} 1 & \text{if } (i,j) \in R \\ 0 & \text{otherwise.} \end{cases}$$

Thus, Equation (1.1) becomes

$$m_{pq} = \sum_{i=1}^{N} \sum_{j=1}^{N} i^p j^q B(i,j). \qquad \text{(Eq 1.2)}$$

In this thesis, we look at computing moments of *binary horizontally/vertically convex* (HV-convex) images. In a horizontally/vertically convex image, any black pixels in a row or column are contiguous. Even though moments become susceptible to noise as the order increases, in most applications in image processing and pattern recognition, the moments used are generally from zero to third order moments [GH]. The computation of central moments, μ_{pq}, uses the moments computed from Equation 1.2:

$$\mu_{pq} = \sum_{i=1}^{N}\sum_{j=1}^{N}\left(i-\bar{x}\right)^{p}\left(j-\bar{y}\right)^{q} f(i,j), \qquad \text{(Eq 1.3)}$$

where $\bar{x} = \frac{m_{10}}{m_{00}}$ and $\bar{y} = \frac{m_{01}}{m_{00}}$.

Variables $\bar{x}$ and $\bar{y}$ are known as centroids. The following shows the central moments of order up to three in terms of centroids and moments [Chung]:

$$\begin{aligned}
\mu_{00} &= m_{00} \\
\mu_{10} &= \mu_{01} = 0 \\
\mu_{20} &= m_{20} - \bar{x}m_{10} \\
\mu_{02} &= m_{02} - \bar{y}m_{01} \\
\mu_{11} &= m_{11} - \bar{y}m_{10} \\
\mu_{30} &= m_{30} - 3\bar{x}m_{20} + 2\bar{x}^{2}m_{10} \\
\mu_{12} &= m_{12} - 2\bar{y}m_{11} - \bar{x}m_{02} + 2\bar{y}^{2}m_{10} \\
\mu_{21} &= m_{21} - 2\bar{x}m_{11} - \bar{y}m_{20} + 2\bar{x}^{2}m_{01} \\
\mu_{03} &= m_{03} - 3\bar{y}m_{02} + 2\bar{y}^{2}m_{01}
\end{aligned}$$

The two most common type of central moments are first moment ($p = q = 1$) and second moment ($p = q = 2$) which corresponds to mean and variance of a particular distribution, respectively. The third central moment ($p = q = 3$) is to compute the

skewness of the distribution (how symmetric or asymmetric the distribution is). The fourth central moment ($p = q = 4$), also known as kurtosis, measures the flatness and peakedness of a distribution [PTVF].

From the central moments, Hu derives a set of seven moments, also known as Hu's moments [COC]. Applications use this set of moments due to its invariant properties. The following shows this set of seven moments.

$$HM_1 = \mu_{20} + \mu_{02}$$

$$HM_2 = (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2$$

$$HM_3 = (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{12})^2$$

$$HM_4 = (\mu_{30} + \mu_{12})^2 + (\mu_{21} + \mu_{03})^2$$

$$\begin{aligned} HM_5 = {} & (\mu_{30} - 3\mu_{12})(\mu_{30} + \mu_{12})\left[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2\right] \\ & + (3\mu_{21} - \mu_{03})(\mu_{21} + \mu_{03})\left[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2\right] \end{aligned}$$

$$HM_6 = (\mu_{20} - \mu_{02})\left[(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2\right] + 4\mu_{11}(\mu_{30} + \mu_{12}) + (\mu_{21} + \mu_{03})$$

$$\begin{aligned} HM_7 = {} & (3\mu_{21} - \mu_{30})(\mu_{30} + \mu_{12})\left[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2\right] \\ & + (3\mu_{12} - \mu_{30})(\mu_{21} + \mu_{03})\left[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2\right] \end{aligned}$$

This set of Hu's moments is just one of the many different types of moments that are being used in pattern recognition. The various types of moments include Zernike moments, pseudo-Zernike moments, Legendre moments, rotational moments, and complex moments [TC].

1.5 Applications of Moments

As previously stated, moments are useful to a wide variety of applications due to their invariant properties [COC, CWH, DBM]. For example, Dudani *et al.* [DBM] used moments to identify 3-dimensional objects. Identifying and classifying 3-D objects into classes can be a challenging task for machines as opposed to human beings. In this case, the authors tried to identify different types of aircraft. A binary image taken from an aircraft will undergo some pre-processing to obtain a clean silhouette and boundary. With these attributes and moments computed from an image, they extracted a set of features. Based on these features, a recognition algorithm identifies the aircraft and estimates its position and orientation in space. In this particular application, they used Hu's set of seven moments.

Another application that uses moments is analyzing pavement images [COC]. Pavement distress surveys are a major component in a pavement management system. Manual surveys, however, can be costly and tedious, labor intensive, and sometimes produce inconsistent results. To overcome this obstacle, image processing computes moments from images of different types of distress in the pavement. The computed values represent features that are used for classification of different types of cracks. Again, this particular application looks at Hu's moments.

Detection of breast cancer is another application that uses moments [CWH]. As with the other two applications, moments extract features from breast cancer biopsy images. A recognition algorithm classifies these features. Based on these biopsy images, physicians can distinguish between benign lesions and various degrees of malignant lesions. This application uses moments of up to fourth order.

1.6 Motivation and Layout of Thesis

As mentioned in the previous section, moments are extremely useful in digital image processing due to their invariant properties, particularly in the area of pattern recognition. Computing moments can be a tedious and time consuming process, however, due to all the multiplications and additions involved. If the image to be processed is of a large size, then moment computations can become the bottleneck of the entire system. Thus, we need to find a way to speed up the computation of moments. This can easily be done on an FPGA chip by utilizing the concept of run-time reconfiguration. We divide a large image into segments for processing, and since the logic inside an FPGA is reprogrammable, it can be reconfigured "on the fly" to process the next segment. This will reduce the cost or area required to compute moments of large sizes. This thesis looks at computing moments of a binary horizontally/vertically convex image using run-time reconfiguration on an FPGA chip.

The remainder of this work is as follows. Chapter 2 presents in detail the definition of a horizontally/vertically convex image. We also look at how restricting Equation 1.2 to binary HV-convex images affects moment computations. Finally, we consider the usefulness of run-time reconfiguration and applications of RTR. In Chapter 3, we will apply the concept of RTR and compute moments on an FPGA chip. Initially, we briefly define the two systems used in computing moments, namely, a non-pipelined system and a pipelined system [Wojko]. We will compare those two systems by time cost and the number of CLBs required by each of the systems. After deciding on the system that will give us the best area/time performance, we will look at the logic units required for moment computations. For an image of size 256×256, we compute the moments by

dividing the image into segments and applying the concept of run-time reconfiguration. We also give a description of carry-save adders and Wallace trees and show how carry-save adders are used to combine multiple outputs (due to division of an image into segments) to produce a final result. Finally, we provide an estimate of the total number of CLBs used in our implementation to compute the moments. Chapter 4 looks at variations of binary HV-convex images and whether RTR also applies in those instances. Finally, we summarize our work in Chapter 5.

Chapter 2

Background on Binary, Horizontally/ Vertically Convex Images and Run-Time Reconfiguration

2.1 Binary, Horizontally/Vertically Convex Images

In this thesis, we are mainly looking at binary, horizontally/vertically convex (*HV-convex*) images. As we know, a binary image consists of white and black pixels and the positions of the black and white pixels are not restricted. A binary HV-convex image, however, constrains the arrangement of pixels so that any black pixels in a row are contiguous and any black pixels in a column are also contiguous. For example, if we look at every row and every column of a binary HV-convex image, black pixels cannot be separated by white pixels. Thus, a row (column) of a binary HV-convex image contains

- contiguous black pixels or
- only 1 black pixel or
- no black pixels.

Below is an example of a 4 × 4 binary HV-convex image.

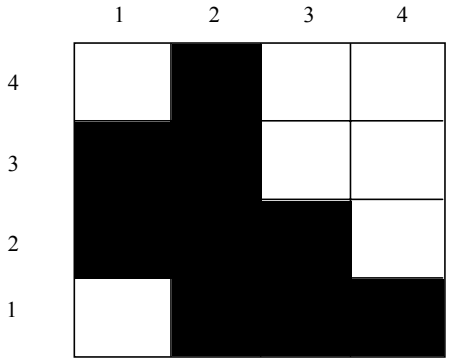

Figure 2.1 4 × 4 binary HV-convex image

2.2 Computing Moments of Binary, Horizontally/Vertically Convex Images on Reconfigurable Meshes

As mentioned in Chapter 1, moment computations using the conventional formula on a large size image involve a lot of multiplications and additions. In particular, computation of Equation 1.1 requires $2N(p+q)$ multiplications and $N \times N$ additions [HCS]. On a binary HV- convex image, however, by taking advantage of how the pixels are arranged, one can compute moments based on the total number of black pixels in a row or column and the position of the leftmost and lowest black pixels [Chung]. A *reconfigurable mesh* (R-Mesh), which consists of processors arranged in an array, can calculate a set of moments in constant time [BPRS]. Each processor has four ports: East (E), West (W), North (N), and South (S). Processors create buses for communication by joining the connections between two distinct ports, such as N-S or E-W. By setting up the port connections, an R-Mesh can compute the number of 1s or 0s in a binary number. For a binary HV-convex image, Chung [Chung] developed an algorithm for an R-Mesh to compute a set of moments that involves determining the total number of contiguous black pixels in each row and each column using the locations of the first and last black pixels.

The moments computed on the R-Mesh are up to 3rd order, for a total of 16 moments, divided into 3 major groups.

Group 1: m_{00}, m_{10}, m_{20}, m_{30}, m_{01}, m_{02}, m_{03}
Group 2: m_{11}, m_{21}, m_{31}, m_{12}, m_{13}
Group 3: m_{22}, m_{23}, m_{32}, m_{33}

The formulas used in computing the moments derive from algebraic manipulation of the conventional formula given in Equation 1.2,

$$m_{pq} = \sum_{i=1}^{N} \sum_{j=1}^{N} i^p j^q B(i, j) .$$

Given that the image is HV-convex, to compute the moments of a binary HV-convex image, Chung has introduced four new terms. Let $\delta^r{}_i$ ($\delta^c{}_i$) denote the length of the string of black pixels in row (column) i, and let $\pi^r{}_i$ ($\pi^c{}_i$) denote the row (column) coordinate of the leftmost (lowest) black pixel in row (column) i. For example, for Figure 2.1, $\delta^r{}_1 = 3$, $\delta^r{}_2 = 3$, $\delta^r{}_3 = 2$, $\delta^r{}_4 = 1$, $\delta^c{}_1 = 2$, $\delta^c{}_2 = 4$, $\delta^c{}_3 = 2$, $\delta^c{}_4 = 1$, $\pi^r{}_1 = 2$, $\pi^r{}_2 = 1$, $\pi^r{}_3 = 1$, $\pi^r{}_4 = 2$, $\pi^c{}_1 = 2$, $\pi^c{}_2 = 1$, $\pi^c{}_3 = 1$, and $\pi^c{}_4 = 1$. In the next section, we will look at how the specialized versions of Equation 1.2 incorporate these new terms to compute moments.

2.3 Comparison between Conventional and Modified Method of Moment Computation

So far, from Chapter 1, moment computation can sometimes be computation intensive due to the number of multiplication and addition operations required. Since Chung specifically considered a binary HV-convex image, he developed equations using the four new terms to compute the moments more efficiently [Chung]. For example, to compute m_{pq} using Equation 1.2, each pixel, $B(i, j)$, will have to be multiplied with i^p and j^q before summing the values of every column for each row, which is, in effect, a 2-D computation. This section will look at how Chung modified Equation 1.2 to reduce 2-D computations to 1-D computations. In general, for an $N \times N$ image, using the modified version of moment computation, the following are the formulas used in computing all 16 moments. We partition the moments into three groups, spreading across several pages.

Group 1:

$$m_{00} = \sum_{i=1}^{N} \delta^r{}_i = \sum_{i=1}^{N} \delta^c{}_i \qquad \text{(Eq 2.1)}$$

$$m_{10} = \sum_{i=1}^{N} \delta^{r}{}_{i} * i \qquad \text{(Eq 2.2)}$$

$$m_{20} = \sum_{i=1}^{N} \delta^{r}{}_{i} * i^{2} \qquad \text{(Eq 2.3)}$$

$$m_{30} = \sum_{i=1}^{N} \delta^{r}{}_{i} * i^{3} \qquad \text{(Eq 2.4)}$$

$$m_{01} = \sum_{i=1}^{N} \delta^{c}{}_{i} * i \qquad \text{(Eq 2.5)}$$

$$m_{02} = \sum_{i=1}^{N} \delta^{c}{}_{i} * i^{2} \qquad \text{(Eq 2.6)}$$

$$m_{03} = \sum_{i=1}^{N} \delta^{c}{}_{i} * i^{3} \qquad \text{(Eq 2.7)}$$

From Equation 1.2, $m_{00} = \sum_{i=1}^{N}\sum_{j=1}^{N} i^{0} j^{0} B(i,j) = \sum_{i=1}^{N}\sum_{j=1}^{N} B(i,j)$, which is basically the summation of all the black pixels in the image. Equation 2.1 obtains m_{00} by summing up the black pixels in a horizontal manner ($\sum_{i=1}^{N} \delta^{r}{}_{i}$) or vertical manner ($\sum_{i=1}^{N} \delta^{c}{}_{i}$). From Equation 1.2, $m_{10} = \sum_{i=1}^{N}\sum_{j=1}^{N} i^{1} j^{0} B(i,j) = \sum_{i=1}^{N}\sum_{j=1}^{N} i * B(i,j)$, which is a 2-D computation. Each pixel is multiplied with its row number. Since this is a binary image where a pixel value can either be 1 or 0, Equation 2.2 multiplies the number of black pixels in a row with the row number. Equation 2.2 reduces the computations for m_{10} to 1-D, $m_{10} = \sum_{i=1}^{N}\sum_{j=1}^{N} i * B(i,j) = \sum_{i=1}^{N} i * \sum_{j=1}^{N} B(i,j) = \sum_{i=1}^{N} \delta^{r}{}_{i} * i$. The same concept applies to both m_{20} and m_{30}. Similarly, when computing moments m_{01}, m_{02}, and m_{03}, Chung looked at the number of black pixels in each column ($\delta^{c}{}_{i}$).

Group 2:

$$m_{11,i} = \delta^{r}{}_{i} * i * \left(\pi^{r}{}_{i} + \frac{\delta^{r}{}_{i} - 1}{2} \right) \qquad \text{(Eq 2.8)}$$

$$m_{11} = \sum_{i=1}^{N} m_{11,i} \qquad \text{(Eq 2.8.1)}$$

$$m_{21,i} = \delta^{r}{}_{i} * i^{2} * \left(\pi^{r}{}_{i} + \frac{\delta^{r}{}_{i} - 1}{2} \right) \qquad \text{(Eq 2.9)}$$

$$m_{21} = \sum_{i=1}^{N} m_{21,i} \qquad \text{(Eq 2.9.1)}$$

$$m_{31,i} = \delta^r{}_i * i^3 * \left(\pi^r{}_i + \frac{\delta^r{}_i - 1}{2} \right) \qquad \text{(Eq 2.10)} \qquad m_{31} = \sum_{i=1}^{N} m_{31,i} \qquad \text{(Eq 2.10.1)}$$

$$m_{12,i} = \delta^c{}_i * i^2 * \left(\pi^c{}_i + \frac{\delta^c{}_i - 1}{2} \right) \qquad \text{(Eq 2.11)} \qquad m_{12} = \sum_{i=1}^{N} m_{12,i} \qquad \text{(Eq 2.11.1)}$$

$$m_{13,i} = \delta^c{}_i * i^3 * \left(\pi^c{}_i + \frac{\delta^c{}_i - 1}{2} \right) \qquad \text{(Eq 2.12)} \qquad m_{13} = \sum_{i=1}^{N} m_{13,i} \qquad \text{(Eq 2.12.1)}$$

We simplify the equations by substituting the following terms

$$Q^r{}_i = \left(\pi^r{}_i + \frac{\delta^r{}_i - 1}{2} \right) \quad \text{and} \quad Q^c{}_i = \left(\pi^c{}_i + \frac{\delta^c{}_i - 1}{2} \right).$$

From Equation 1.2, $m_{11} = \sum_{i=1}^{N} \sum_{j=1}^{N} i^1 j^1 B(i,j)$. With the combination of leftmost pixel ($\pi^r{}_i$) and length of black pixels in a row ($\delta^r{}_i$), however, Chung obtained Equation 2.8 as follows:

$$m_{11,i} = \pi^r{}_i * i + \left(\pi^r{}_i + 1 \right) * i + ... + \left(\pi^r{}_i + \delta^r{}_i - 1 \right) * i$$

$$= \delta^r{}_i * \pi^r{}_i * i + \frac{\left(\delta^r{}_i - 1 \right) * \delta^r{}_i * i}{2}$$

$$= \delta^r{}_i * i * \left(\pi^r{}_i + \frac{\delta^r{}_i - 1}{2} \right)$$

By summing up $m_{11,i}$ for $1 \le i \le N$, Chung calculated moment m_{11}, as shown by Equation 2.8.1. Moments m_{21} and m_{31} follow along similar lines, as shown by Equations 2.9.1 and 2.10.1, respectively. Compute Equations 2.11 and 2.12 by swapping $\pi^r{}_i$ with $\pi^c{}_i$ and $\delta^r{}_i$ with $\delta^c{}_i$.

$$m_{1y,i} = \pi^c{}_i * i^y + \left(\pi^c{}_i + 1 \right) * i^y + ... + \left(\pi^c{}_i + \delta^c{}_i - 1 \right) * i^y$$

$$= \delta^c{}_i * \pi^c{}_i * i^y + \frac{\left(\delta^c{}_i - 1\right) * \delta^c{}_i * i^y}{2}$$

$$= \delta^c{}_i * i^y * \left(\pi^c{}_i + \frac{\delta^c{}_i - 1}{2}\right) \qquad \text{for } 2 \le y \le 3$$

Group 3:

$$m_{22,i} = i^2 * \left[\delta^r{}_i * \left(\pi^r{}_i\right)^2 + \frac{\left(\delta^r{}_i - 1\right) * \delta^r{}_i * \left(2 * \delta^r{}_i - 1\right)}{6} + \pi^r{}_i * \left(\delta^r{}_i - 1\right) * \delta^r{}_i\right] \qquad \text{(Eq 2.13)}$$

$$m_{22} = \sum_{i=1}^{N} m_{22,i} \qquad \text{(Eq 2.13.1)}$$

$$m_{32,i} = i^3 * \left[\delta^r{}_i * \left(\pi^r{}_i\right)^2 + \frac{\left(\delta^r{}_i - 1\right) * \delta^r{}_i * \left(2 * \delta^r{}_i - 1\right)}{6} + \pi^r{}_i * \left(\delta^r{}_i - 1\right) * \delta^r{}_i\right] \qquad \text{(Eq 2.14)}$$

$$m_{32} = \sum_{i=1}^{N} m_{32,i} \qquad \text{(Eq 2.14.1)}$$

$$m_{23,i} = i^3 * \left[\delta^c{}_i * \left(\pi^c{}_i\right)^2 + \frac{\left(\delta^c{}_i - 1\right) * \delta^c{}_i * \left(2 * \delta^c{}_i - 1\right)}{6} + \pi^c{}_i * \left(\delta^c{}_i - 1\right) * \delta^c{}_i\right] \qquad \text{(Eq 2.15)}$$

$$m_{23} = \sum_{i=1}^{N} m_{23,i} \qquad \text{(Eq 2.15.1)}$$

$$m_{33,i} = i^3 * \left[\left(\pi^c{}_i\right)^3 * \delta^c{}_i + \left(\frac{\left(\delta^c{}_i - 1\right) * \delta^c{}_i}{2}\right)^2 + 3 * \left(\left(\pi^c{}_i\right)^2 * \frac{\left(\delta^c{}_i - 1\right) * \delta^c{}_i}{2}\right) + \pi^c{}_i * \frac{\left(\delta^c{}_i - 1\right) * \delta^c{}_i * \left(2 * \delta^c{}_i - 1\right)}{6}\right] \qquad \text{(Eq 2.16)}$$

$$m_{33} = \sum_{i=1}^{N} m_{33,i} \qquad \text{(Eq 2.16.1)}$$

To further simplify the equations in Group 3 for better representation, let

$$Y^r{}_i = \left(\pi^r{}_i\right)^2 + \frac{\left(\delta^r{}_i - 1\right) * \left(2 * \delta^r{}_i - 1\right)}{6} + \pi^r{}_i * \left(\delta^r{}_i - 1\right), \qquad \text{(Eq 2.17)}$$

$$Y^c{}_i = \left(\pi^c{}_i\right)^2 + \frac{\left(\delta^c{}_i - 1\right) * \left(2 * \delta^c{}_i - 1\right)}{6} + \pi^c{}_i * \left(\delta^c{}_i - 1\right), \text{ and} \quad \text{(Eq 2.18)}$$

$$Z^c{}_i = \left(\pi^c{}_i\right)^3 + \frac{\left(\delta^c{}_i - 1\right)^2 * \delta^c{}_i}{4} + \frac{3 * \left(\pi^c{}_i\right)^2 * \left(\delta^c{}_i - 1\right)}{2} + \frac{\pi^c{}_i * \left(\delta^c{}_i - 1\right) * \left(2 * \delta^c{}_i - 1\right)}{6}. \quad \text{(Eq 2.19)}$$

With the three new terms, the following are the equations contributing to the last four moments.

$$m_{22,i} = i^2 * \delta^r{}_i * Y^r{}_i$$

$$m_{32,i} = i^3 * \delta^r{}_i * Y^r{}_i$$

$$m_{23,i} = i^2 * \delta^c{}_i * Y^c{}_i$$

$$m_{33,i} = i^3 * \delta^c{}_i * Z^r{}_i$$

The following are the 16 moments for a 4 × 4 binary HV-convex image, Figure 2.1.

Group 1:

$m_{00} = 9$

$m_{10} = 19$ $\qquad$ $m_{01} = 20$

$m_{20} = 49$ $\qquad$ $m_{02} = 52$

$m_{30} = 145$ $\qquad$ $m_{03} = 152$

Group 2:

$m_{11} = 38$

$m_{21} = 92$ $\qquad$ $m_{12} = 88$

$m_{31} = 266$ $\qquad$ $m_{13} = 230$

Group 3:

$m_{22} = 194$ $m_{23} = 804$

$m_{32} = 532$ $m_{33} = 832$

2.4 Run-Time Reconfiguration

Run-time reconfiguration (RTR) is an effective technique for FPGA circuits. RTR is basically an approach that divides an application into portions that can be computed as separate configurations. The ability to reconfigure enables certain applications to achieve high performance with a low hardware resource requirement. RTR is useful in the following two instances:

- embedding a large, special-purpose computing architecture onto limited resources, and
- presence of idle or inactive hardware.

In the first case, instead of using a limited size, static architecture to perform computational variations found within an algorithm, partition the algorithm into special-purpose circuits that can be reconfigured on the fly. One example that uses this approach is an image coding system [SJV]. In the second case, RTR reduces the hardware resources for a particular system that may have idle or underutilized hardware. For example, consider a system that is divided into sections that depend on each other. This dependency renders the sections idle or inactive until computations from previous sections have completed. RTR can alter the circuitry by replacing the idle hardware with more useful resources or eliminating the idle hardware and configuring the active hardware for different sections. Instead of staying idle, these resources improve the performance of the computations in the active stage.

One particular application that demonstrates this approach is artificial neural networks [EH]. The run-time reconfigured neural network (RRANN) implements the well-known backpropagation learning algorithm using FPGAs. Backpropagation is a technique used to train node weights within a neural network [Fausett]. As shown in Figure 2.2, this algorithm consists of three stages: feed-forward, backpropagation, and update.

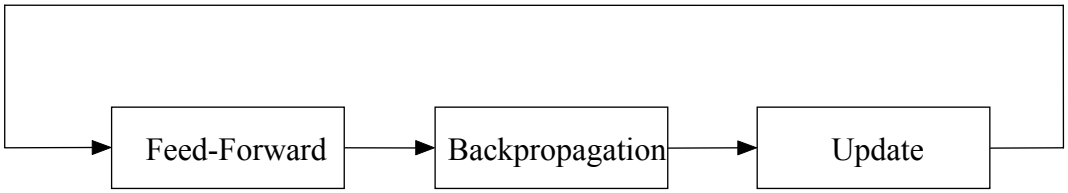

Figure 2.2 Three stages of the backpropagation training algorithm

Each stage is dependent on the outputs produced from the previous stage. For example, the backpropagation stage cannot proceed until calculation in the feed-forward stage have completed. This dependency prevents all three stages from executing in parallel, resulting in two out of three stages being idle or inactive at any time. As shown in Figure 2.3, by using RTR, one can configure FPGA resources specifically for a particular stage. Upon completion of a stage of computations, reconfigure the hardware for the following stage. This process of execution and reconfiguration continues until the training algorithm converges. In this analysis, applying RTR and eliminating the idle circuitry provides a 500% improvement in functional density of the neural network.

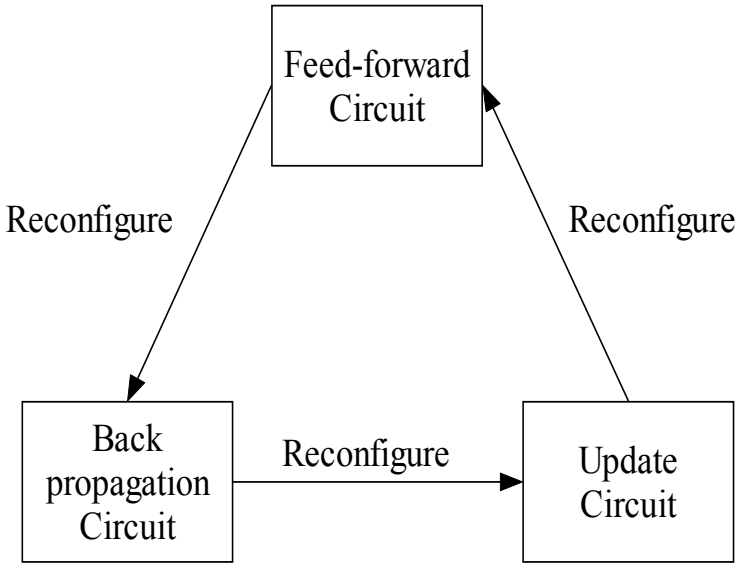

Figure 2.3 Run-time reconfigured neural network (RRANN)

In this thesis, we are using run-time reconfiguration to compute moments of a binary HV-convex image using the formulas in Section 2.2 on an FPGA chip. One logic unit that we will look at in detail in Chapter 3 is the *constant coefficient multiplier* (KCM). A KCM generally multiplies an N-bit wide variable with an N-bit fixed (constant) coefficient to produce a $2N$-bit result. In this thesis, however, we are looking at a multiplier and a multiplicand of different sizes. Thus, multiplication between an m-bit wide multiplicand and an n-bit wide fixed (constant) multiplier will produce an $(m+n)$-bit product.

The FPGA uses ROM-based *look-up tables* (LUTs) to store the precomputed products for every possible value of the multiplicand. The circuit accesses the look-up tables at run-time to provide the required output. Run-time reconfiguration updates the

constant multiplier at run-time or on the fly. A KCM is both smaller and faster than a conventional multiplier when implementing both in an FPGA.

One particular application that uses constant coefficient multipliers and run-time reconfiguration is IDEA encryption [WE]. IDEA is an encryption algorithm that uses a 128-bit length key to encrypt successive 64-bit blocks of plaintext. In general, the algorithm divides a 64-bit block of plaintext into four 16-bit sub-blocks: X1, X2, X3, and X4, which become the input blocks to the first round of the algorithm. Each round XORs, adds, and multiplies the four sub-blocks with six 16-bit sub-key sequences obtained from the 128-bit key. This process will repeat for eight rounds and all six 16-bit sub-keys remain constant during each round of computation.

In each round of computation, however, the multiplication operation uses only four out of six 16-bit sub-keys. Thus, this algorithm requires four KCMs and each 16-bit sub-key is the constant multiplier. After each round, the KCMs have to reconfigure to obtain a new set of 16-bit sub-keys. Each reconfiguration of the KCMs requires 16 clock cycles and each round of computation requires 19 clock cycles. This will allow the KCMs sufficient time to reconfigure for the next set of 16-bit sub-keys.

2.5 Conclusion

Chung has developed a set of new formulas to compute moments of a binary horizontally/vertically convex image, which are less complex than Equation 1.2. These new formulas involve multiplications with a constant variable. In order to speed up the calculations, we can utilize constant coefficient multipliers that are available in FPGAs. Furthermore, run-time reconfiguration is a useful technique in optimizing the circuits for a particular application. By dividing a problem into segments, we can use a set of

hardware to process all segments by reconfiguring the hardware before processing the next segment. In the next chapter, we will look at computing moments of a binary HV-convex image on an FPGA using run-time reconfiguration.

Chapter 3

Computing Moments of Binary and Horizontally/Vertically Convex Images Using Run-Time Reconfiguration

3.1 Applying the Concept of Run-Time Reconfiguration

In this thesis, we are looking into computing moments for a 256 × 256 binary HV-convex image using an FPGA. Processing the whole image at the same time will require a large amount of resources. Thus, we are looking into dividing the image into certain portions and processing each portion separately by utilizing the concept of run-time reconfiguration available to FPGAs.

As we can see, all the 16 equations for the moments m_{00} through m_{33} involve a lot of multiplications and additions. With further scrutiny and slight modifications, we notice that some of the equations have a common term and through this we will try to find an easier and faster way of computing the moments. By finding a common term used in multiplications, we can use a constant coefficient multiplier (KCM) instead of a general multiplier to produce the results. Recall that a KCM can be both smaller and faster than a general multiplier.

The 16 equations possess one common recurring theme: the value 'i' is used repeatedly as a multiplier. For example from Group 1,

$$\delta^r{}_i \xrightarrow{*\,i} i*\delta^r{}_i \xrightarrow{*\,i} i^2*\delta^r{}_i \xrightarrow{*\,i} i^3*\delta^r{}_i .$$

We take advantage of this repetition to use a constant coefficient multiplier configured with constant value *i* to obtain the moments. In this thesis, we define a *module* as a collection of logic units such as constant coefficient multipliers, general multipliers, and adders. Each module processes one whole column and one whole row of the image, and the constant multiplier, *i*, corresponds to the column number and row number. Computations that involve *i* use constant coefficient multipliers, as shown in Table 3.1. Multiplications between variables other than *i*, including outputs from KCMs, use general multipliers. Furthermore, Equations 2.17, 2.18, and 2.19 show an example of the use of adders to compute $Y^r{}_i$, $Y^c{}_i$, and $Z^c{}_i$. Since each module processes only one column and one row of an image, this thesis looks at using multiple modules on an FPGA chip to fully utilize the available resources on the chip. Each module produces terms for moment computations. From the equations in all three groups from Chapter 2, we compute moments by summing up the terms of every column or every row, depending on the type of moment. Thus, we need to add the output produced by each module that corresponds to that particular moment. Depending on the number of modules inside an FPGA and the size of the image, this process repeats itself. For example, if an FPGA chip embeds eight modules and the image size is 256 × 256, then we have to repeat the process for 256/8 = 32 times. Besides having to keep track of the additions needed within the eight modules, we also have to take into consideration the sums from previous sets of eight modules.

To provide a better explanation, we will initially focus our attention on only one module. The two basic components in a module are a general multiplier and a constant coefficient multiplier. As stated above, each module will process one column and one row at a time. Along with other logic units such as multiplexers and adders, both the

KCM and the general multiplier will produce terms that are required for moment computation. The KCM will reconfigure itself on the fly for the next value of *i*. The reconfiguration of the KCM will take 16 clock cycles [DCCM]. In later sections, we will look in detail at how the terms are produced and summations of these terms across modules. This design looks at two different cases:

- non-pipelined and
- pipelined.

In both cases, we will look at different combinations of general multipliers and constant coefficient multipliers.

- 1 general multiplier, 1 KCM
- 1 general multiplier, 2 KCMs
- 2 general multipliers, 1 KCM
- 2 general multipliers, 2 KCMs

Table 3.1 states specifically which terms are being produced by general multipliers and KCMs.

Table 3.1 Terms computed by general multiplier and constant coefficient multiplier

Constant Coefficient Multiplier	General Multiplier
$i * \delta^r_i$ $i^2 * \delta^r_i$ $i^3 * \delta^r_i$ $i * \delta^c_i$ $i^2 * \delta^c_i$ $i^3 * \delta^c_i$ $i * \delta^r_i * Q^r_i$ $i^2 * \delta^r_i * Q^r_i$ $i^3 * \delta^r_i * Q^r_i$ $i^3 * \delta^r_i * Y^r_i$ $i^3 * \delta^c_i * Q^c_i$	$\delta^r_i * Q^r_i$ $i^2 * \delta^r_i * Y^r_i$ $i^2 * \delta^c_i * Q^c_i$ $i^3 * \delta^c_i * Y^c_i$ $i^3 * \delta^c_i * Z^c_i$

Figure 3.1 shows a rough sketch of the two connections between the basic components in a module.

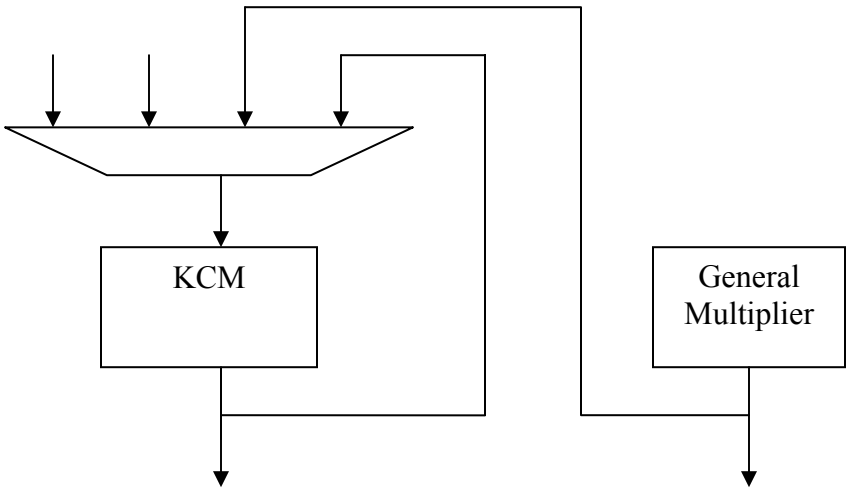

Figure 3.1 Interconnections between general multiplier and constant coefficient multiplier

3.2 Computations in a Single Module

As stated in the previous section, first we are concentrating on the computations required in a single module. We are looking specifically at two cases:

- non-pipelined system and
- pipelined system.

In the next few subsections, we will consider the logic gates needed in the module.

3.2.1 Comparison between Non-Pipelined and Pipelined Systems

In this subsection, we are trying to see whether a non-pipelined or a pipelined system is a better approach in computing the moments for a binary HV-convex image. A pipelined system divides operations into stages that execute in parallel. A set of inputs will move through every stage before producing the final result. Thus, a pipelined system

will be able to accept a continuous stream of inputs without waiting for the completion of the previous set of inputs. A non-pipelined system, however, mandates the completion of the current set of inputs before processing the next set of inputs. We will compare both non-pipelined and pipelined systems by looking at the time cost and the number of CLBs required by each system to compute the moments. The main purpose of this subsection is to determine the combination of constant coefficient multipliers and general multipliers that best suits this design, specifically in terms of the total time taken to compute the moments.

3.2.1.1 Non-pipelined Constant Coefficient Multipliers and General Multipliers

For the non-pipelined case, we are trying to schedule the multiplications according to the number of available multipliers (four different combinations of KCMs and general multipliers) with the following assumptions:

- a KCM is twice as fast as a general multiplier, and
- values of δ's, π's, and Q's have been pre-computed.

Xilinx application notes [CCM, VM] give the number of CLBs required for constant coefficient multipliers and general multipliers with inputs of equal sizes. As shown in Figures 3.2 and 3.3, a multiplier with two different sized inputs will also have approximately similar number of CLBs, as long as they have the same output size. For example, a 32 × 8 multiplier will have approximately the same number of CLBs as a 20 × 20 multiplier since both of them will produce a 40-bit output. In Figure 3.2, a 20 × 20 multiplier uses five 16 × 24 LUTs, three 24-bit adders (Adders 1 and 3), and one 29-bit adder (Adder 2). In Figure 3.3, a 32 × 8 multiplier uses eight 16 × 12 LUTs, four 12-bit adders (Adder 4), one 16-bit adder (Adder 5), one 17-bit adder (Adder 6), and one 24-bit

adder (Adder 7). Note that a $k \times l$ LUT is a LUT with k entries, each of l-bits wide. An m-bit adder adds two m-bit inputs and produces an (m+1)-bit output. The number of CLBs for one 16 × 24 LUT and one 24-bit adder is twice the number of CLBs for one 16 × 12 LUT and one 12-bit adder, respectively. Thus, in terms of 16 × 12 LUTs, a 20 × 20 multiplier consumes approximately ten 16 × 12 LUTs. From Figure 3.3, a 32 × 8 multiplier uses only eight 16 × 12 LUTs. A 32 × 8 multiplier, however, needs more CLBs for its additional adders, namely Adder 5 and Adder 6. In conclusion, a 20 × 20 multiplier and a 32 × 8 multiplier use approximately the same number of CLBs. In both Figures 3.2 and 3.3, p:q denotes the start bit, p and end bit, q. (q-p)+1 represents the total bits.

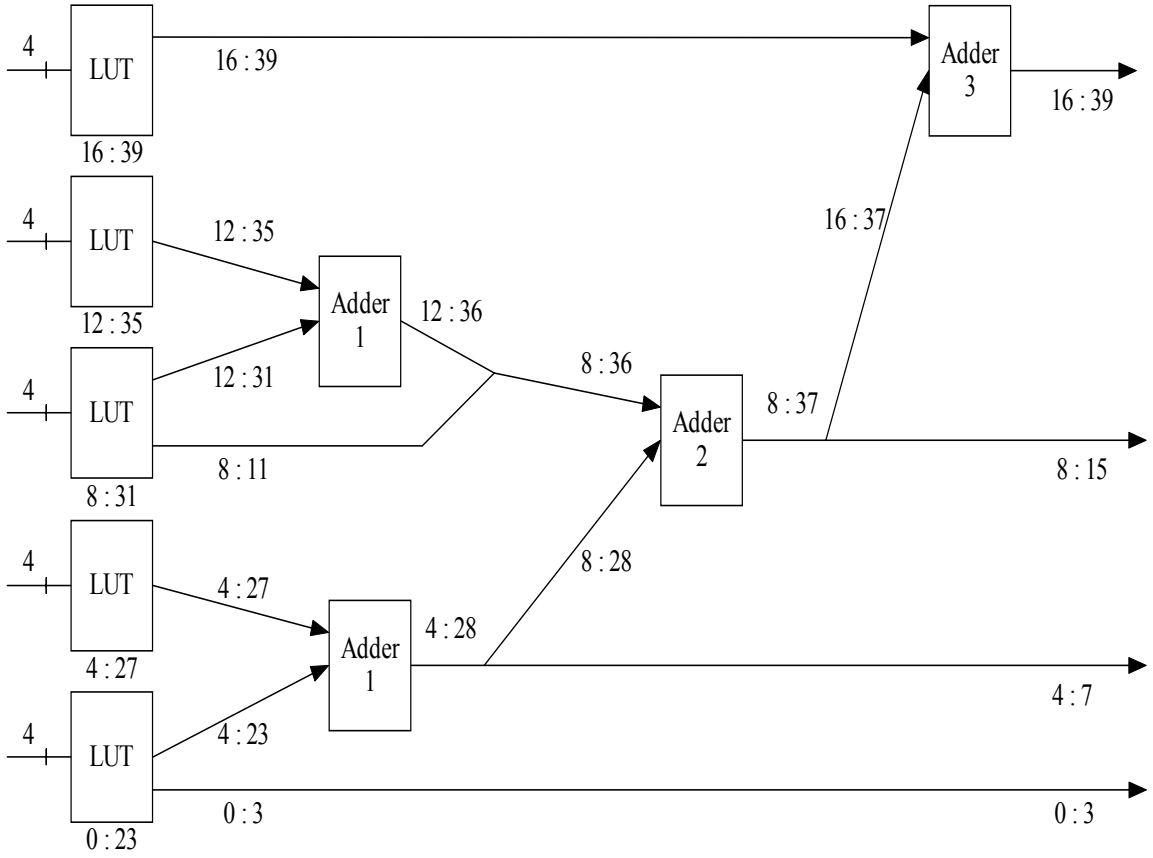

Figure 3.2 20 × 20 multiplier

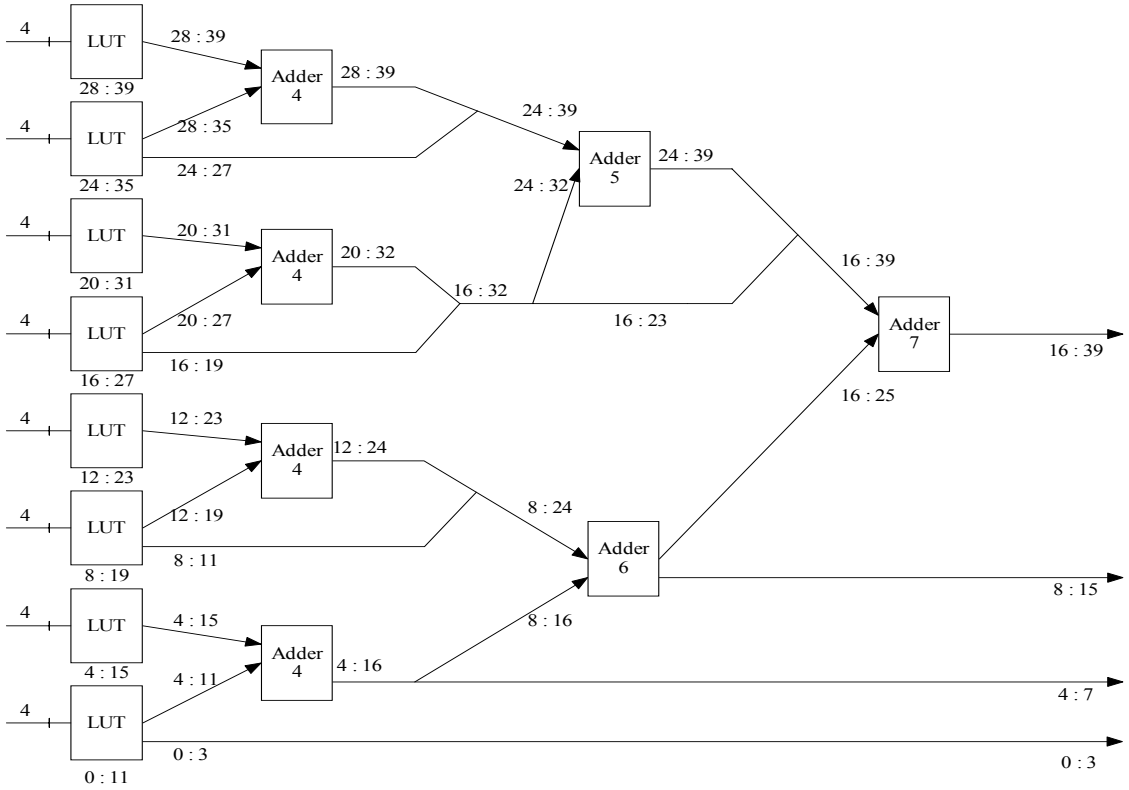

Figure 3.3 32 × 8 multiplier

In our design, the largest outputs produced by a constant coefficient multiplier and a general multiplier are 54 bits ($i^3 * \delta^r{}_i * Y^r{}_i$) and 64 bits ($i^3 * \delta^c{}_i * Z^c{}_i$), respectively. From a Xilinx application note [CCM], non-pipelined KCMs producing 24-bit and 32-bit outputs use approximately 25 CLBs (10 rows, 3 columns) and 43 CLBs (12 rows, 4 columns) respectively. As shown above, the actual CLB count is approximately five CLBs less than the CLB array size (multiplying number of rows and number of columns). Roughly obtain the number of slices by multiplying the actual CLB count by 1.7. For every 8-bit increment, the number of rows increases by two and the number of columns increases by one. Thus, a non-pipelined KCM producing a 56-bit output, which is close to 54 bits, will occupy approximately 121 CLBs (18 rows, 7 columns) or 206 slices (1.7 * 121). By using the same comparison, non-pipelined general multipliers

producing 24-bit and 32-bit outputs use approximately 49 CLBs (9 rows, 6 columns) and 91 CLBs (12 rows, 8 columns), respectively. For every 8-bit increment, the number of rows increases by three and the number of columns increases by two. Thus, a non-pipelined general multiplier producing 64 bits consumes 379 CLBs (24 rows, 16 columns) or 645 slices (1.7 * 379). The first scenario with one KCM and one general multiplier, choosing 16 as the total number of modules (since 256 is divisible by 16), uses approximately 500 CLBs. This number will be a reference point for the other three cases. Table 3.2 shows how many modules are used to process the whole 256 × 256 image and the number of iterations required.

Table 3.2 Number of modules needed for all four cases without pipelining

	Number of CLBs for multipliers per module	Total number of modules	Total CLBs for multipliers	Number of iterations needed
1 KCM and 1 general multiplier	(1 * 121) + (1 * 379) = 500	16	8000	16
1 KCM and 2 general multipliers	(1 * 121) + (2 * 379) = 879	9	7911 ≈ 8000	29
2 KCMs and 1 general multiplier	(2 * 121) + (1 * 379) = 621	13	8073 ≈ 8000	20
2 KCMs and 2 general multipliers	(2 * 121) + (2 * 379) = 1000	8	8000	32

The next four tables will show the schedule of operations in the multipliers in all four cases. To be compact regarding which terms are involved in the multiplications, we have represented some of the terms with new variables as shown here.

$A = (\delta^r_i - 1)$ $\quad B = (2 * \delta^r_i - 1)$ $\quad C = (\delta^c_i - 1)$
$D = (2 * \delta^c_i - 1)$ $\quad E = \pi^c_i * C$ $\quad F = 1/6 * C * D$

In the cases where there are two constant coefficient multipliers, Tables 3.3 and 3.5, the values used as constants are i and π^c_i. In all four scenarios, the main purpose of

scheduling the multiplications is to find the least number of clock cycles required to produce the results, including 16 clock cycles needed by a KCM for reconfiguration.

Note that the scheduling employs the assumption that multiplications in a KCM take one clock cycle and multiplications using general multipliers take two clock cycles. In all four tables, the notation '-' denotes a busy clock cycle while an empty box represents an idle clock cycle. The total number of clock cycles in each module is:

max{total clock cycles in KCM + reconfiguration time, total clock cycles in general multiplier}.

Table 3.3 Schedule for module with two KCMs and two general multipliers without pipelining

	KCM	General Multiplier	General Multiplier	KCM
1	$i * \delta^r_i$	$\delta^r_i * Q^r_i$	$\pi^r_i * \pi^r_i$ (part of Y^r_i)	$\pi^c_i * \pi^c_i$ (part of Y^c_i)
2	$i^2 * \delta^r_i$	-	-	$\pi^c_i * C$ (part of Y^c_i)
3	$i^3 * \delta^r_i$	$C * D$ (part of Y^c_i)	$A * B$ (part of Y^r_i)	$\pi^c_i * \pi^{c2}_i$ (part of Z^c_i)
4	$i * \delta^c_i$	-	-	$\pi^c_i * E$ (part of Z^c_i)
5	$i^2 * \delta^c_i$	1/6 *(C*D) (part of Y^c_i)	1/6 *(A*B) (part of Y^r_i)	
6	$i^3 * \delta^c_i$	-	-	
7	$i * \delta^r_i * Q^r_i$	$i^2 * \delta^c_i * Q^c_i$	$\pi^r_i * A$ (part of Y^r_i)	
8	$i^2 * \delta^r_i * Q^r_i$	-	-	
9	$i^3 * \delta^r_i * Q^r_i$	$i^3 * \delta^c_i * Y^c_i$	$i^2 * \delta^r_i * Y^r_i$	
10		-	-	
11	$i^3 * \delta^c_i * Q^c_i$	3* (π^c_i*E) (part of Z^c_i)	$C * C$ (part of Z^c_i)	
12	$i^3 * \delta^r_i * Y^r_i$	-	-	
13			$\delta^c_i * C^2$ (part of Z^c_i)	
14			-	
15			$\pi^c_i * F$ (part of Z^c_i)	
16			-	
17			$i^3 * \delta^c_i * Z^c_i$	
18			-	

Table 3.4 Schedule for module with one KCM and one general multiplier without pipelining

	KCM	General Multiplier
1	$i * \delta^r_i$	$\delta^r_i * Q^r_i$
2	$i^2 * \delta^r_i$	-
3	$i^3 * \delta^r_i$	$\pi^r_i * \pi^r_i$ (part of Y^r_i)
4	$i * \delta^c_i$	-
5	$i^2 * \delta^c_i$	$A * B$ (part of Y^r_i)
6	$i^3 * \delta^c_i$	-
7	$i * \delta^r_i * Q^r_i$	1/6 *(A*B) (part of Y^r_i)
8	$i^2 * \delta^r_i * Q^r_i$	-
9	$i^3 * \delta^r_i * Q^r_i$	$\pi^r_i * A$ (part of Y^r_i)
10		-
11		$i^2 * \delta^r_i * Y^r_i$
12		-
13	$i^3 * \delta^r_i * Y^r_i$	$i^2 * \delta^c_i * Q^c_i$
14		-
15	$i^3 * \delta^c_i * Q^c_i$	$\pi^c_i * \pi^c_i$ (part of Y^c_i)
16		-
17		$C * D$ (part of Y^c_i)
18		-
19		1/6 *(C*D) (part of Y^c_i)
20		-
21		$\pi^c_i * C$ (part of Y^c_i)
22		-
23		$i^3 * \delta^c_i * Y^c_i$
24		-
25		$\pi^c_i * (\pi^c_i)^2$ (part of Z^c_i)
26		-
27		$C * C$ (part of Z^c_i)
28		-
29		$\delta^c_i * C^2$ (part of Z^c_i)
30		-
31		$\pi^c_i * E$ (part of Z^c_i)
32		-
33		3* (π^c_i*E) (part of Z^c_i)
34		-
35		$\pi^c_i * F$ (part of Z^c_i)
36		-
37		$i^3 * \delta^c_i * Z^c_i$
38		-

Table 3.5 Schedule for module with two KCMs and one general multiplier without pipelining

	KCM	General Multiplier	KCM
1	$i * \delta^r_i$	$\delta^r_i * Q^r_i$	$\pi^c_i * \pi^c_i$ (part of Y^c_i)
2	$i^2 * \delta^r_i$	-	$\pi^c_i * C$ (part of Y^c_i)
3	$i^3 * \delta^r_i$	$\pi^r_i * \pi^r_i$ (part of Y^r_i)	$\pi^c_i * \pi^{c\,2}_i$ (part of Z^c_i)
4	$i * \delta^c_i$	-	$\pi^c_i * E$ (part of Z^c_i)
5	$i^2 * \delta^c_i$	$A * B$ (part of Y^r_i)	
6	$i^3 * \delta^c_i$	-	
7	$i * \delta^r_i * Q^r_i$	1/6 *(A*B) (part of Y^r_i)	
8	$i^2 * \delta^r_i * Q^r_i$	-	
9	$i^3 * \delta^r_i * Q^r_i$	$\pi^r_i * A$ (part of Y^r_i)	
10		-	
11		$i^2 * \delta^r_i * Y^r_i$	
12		-	
13	$i^3 * \delta^r_i * Y^r_i$	$i^2 * \delta^c_i * Q^c_i$	
14		-	
15	$i^3 * \delta^c_i * Q^c_i$	$C * D$ (part of Y^c_i)	
16		-	
17		1/6 *(C*D) (part of Y^c_i)	
18		-	
19		$i^3 * \delta^c_i * Y^c_i$	
20		-	
21		$C * C$ (part of Z^c_i)	
22		-	
23		$\delta^c_i * C^2$ (part of Z^c_i)	
24		-	
25		3* (π^c_i*E) (part of Z^c_i)	
26		-	
27		$\pi^c_i * F$ (part of Z^c_i)	
28		-	
29		$i^3 * \delta^c_i * Z^c_i$	
30		-	

Table 3.6 Schedule for module with one KCM and two general multipliers without pipelining

	KCM	General Multiplier	General Multiplier
1	$i * \delta^r_i$	$\delta^r_i * Q^r_i$	$\pi^r_i * \pi^r_i$ (part of Y^r_i)
2	$i^2 * \delta^r_i$	-	-
3	$i^3 * \delta^r_i$	$\pi^c_i * \pi^c_i$ (part of Y^c_i)	$A * B$ (part of Y^r_i)
4	$i * \delta^c_i$	-	-
5	$i^2 * \delta^c_i$	$C * D$ (part of Y^c_i)	1/6 *(A*B) (part of Y^r_i)
6	$i^3 * \delta^c_i$	-	-
7	$i * \delta^r_i * Q^r_i$	1/6 *(C*D) (part of Y^c_i)	$\pi^r_i * A$ (part of Y^r_i)
8	$i^2 * \delta^r_i * Q^r_i$	-	-
9	$i^3 * \delta^r_i * Q^r_i$	$\pi^c_i * C$ (part of Y^c_i)	$i^2 * \delta^r_i * Y^r_i$
10		-	-
11	$i^3 * \delta^r_i * Y^r_i$	$i^3 * \delta^c_i * Y^c_i$	$i^2 * \delta^c_i * Q^c_i$
12		-	-
13	$i^3 * \delta^c_i * Q^c_i$	$\pi^c_i * (\pi^c_i)^2$ (part of Z^c_i)	$\pi^c_i * E$ (part of Z^c_i)
14		-	-
15		$C * C$ (part of Z^c_i)	3* (π^c_i*E) (part of Z^c_i)
16		-	-
17		$\delta^c_i * C^2$ (part of Z^c_i)	$\pi^c_i * F$ (part of Z^c_i)
18		-	-
19		$i^3 * \delta^c_i * Z^c_i$	
20		-	

Table 3.7 compares the number of clock cycles needed to produce the outputs needed for computing moments of an image.

Table 3.7 Total clock cycles without pipelining

	# of clock cycles needed for each iteration	# of iterations	Total # of clock cycles for all iterations
1 KCM and 1 general multiplier	max{15+16,38} = 38	16	16*38 = 608
1 KCM and 2 general multipliers	max{13+16,20} = 29	29	29*29 = 841
2 KCMs and 1 general multiplier	max{15+16,30} = 31	20	20*31 = 620
2 KCMs and 2 general multipliers	max{12+16,18} = 28	32	32*28 = 896

As shown in Table 3.7, even though initially the case with just one KCM and one general multiplier seems to be the slowest at 38 clock cycles, with 16 modules in an

FPGA chip, the total number of clock cycles needed is only 608. The scenario with two KCMs and one general multiplier has 620 total clock cycles but at the cost of one extra KCM (121 CLBs). Besides that, looking at Tables 3.3 and 3.5 where both have two KCMs, the second KCM with π^c_i as a constant is not fully utilized, being occupied only for the first four cycles and stays idle for the rest of the operation. Thus, the best combination is one KCM and one general multiplier.

3.2.1.2 Pipelined Constant Coefficient Multipliers and General Multipliers

As with the case of non-pipelined multipliers, we are still looking at scheduling multiplications in all four scenarios with the following assumptions:

- a KCM is approximately twice as fast as a general multiplier, and
- values of δ's, π's, and Q's have been pre-computed.

In this design, a pipelined KCM and a pipelined general multiplier will consist of four and six stages, respectively. Each multiplication in a KCM will take four clock cycles and a multiplication in a general multiplier will take six clock cycles. The estimated numbers of CLBs needed for a KCM and a general multiplier are again based on Xilinx application notes [DCCM, VM]. The largest outputs produced by a constant coefficient multiplier and a general multiplier are 54 bits ($i^3 * \delta^r{}_i * Y^r{}_i$) and 64 bits ($i^3 * \delta^c{}_i * Z^c{}_i$), respectively. From a Xilinx application note [CCM], there is a difference of about ten slices between a non-pipelined KCM and a pipelined KCM with three stages. For example, there is a difference of 12 slices for a three stage pipelined KCM and a non-pipelined KCM, both producing a 24-bit output and a difference of eight slices between a three stage pipelined KCM and a non-pipelined KCM, both producing a 32-bit output. As discussed in Section 3.2.1.1, a non-pipelined KCM producing a 56-bit output uses approximately 206 slices. Since the more stages a pipelined KCM has, the more

slices it uses, a pipelined KCM with four stages producing a 56-bit output consumes about 221 slices (206 + 15) or 124 CLBs. From a Xilinx application note [VM], a pipelined general multiplier with five stages has 25 slices more than a non-pipelined general multiplier. Again, from Section 3.2.1.1, a non-pipelined general multiplier producing a 64-bit output uses 645 slices. Thus, a pipelined general multiplier with six stages requires approximately 675 slices (645 + 30) or 382 CLBs. In the case with one constant coefficient multiplier and one general multiplier, the total number of CLBs needed for multipliers in all 16 modules is 8096, which again will be used as the reference point. Table 3.8 shows the number of CLBs needed for all four cases.

Table 3.8 Number of modules needed for all four cases with pipelining

	Number of CLBs for multipliers per module	Total number of modules	Total CLBs for multipliers	Number of iterations needed
1 KCM and 1 general multiplier	(1 * 124) + (1 * 382) = 506	16	8096	16
1 KCM and 2 general multipliers	(1 * 124) + (2 * 382) = 888	9	7992 ≈ 8096	29
2 KCMs and 1 general multiplier	(2 * 124) + (1 * 382) = 630	13	8190 ≈ 8096	20
2 KCMs and 2 general multipliers	(2 * 124) + (2 * 382) = 1012	8	8096	32

Similar to the non-pipelined scenario, the next four tables will show the schedule of each of the multiplications in all the different cases with some terms substituted with the following variables.

$A = (\delta^{r}_{i} - 1)$ $B = (2 * \delta^{r}_{i} - 1)$ $C = (\delta^{c}_{i} - 1)$
$D = (2 * \delta^{c}_{i} - 1)$ $E = \pi^{c}_{i} * C$ $F = 1/6 * C * D$

Note that in all four tables, a pipelined KCM and a pipelined general multiplier have four

and six stages, respectively. To fit the tables in a single page, each table will only show active stages. For example, in Table 3.12, there are only three and four active stages in the general multipliers.

Table 3.9 Schedule for module with one KCM and one general multiplier with pipelining

	KCM				General Multiplier					
1										
2	$i * \delta^r_i$				$\delta^r_i * Q^r_i$					
3		$i * \delta^c_i$				$A * B$ (part of Y^r_i)				
4							$\pi^r_i * \pi^r_i$ (part of Y^r_i)			
5	$i^2 * \delta^r_i$							$\pi^r_i * A$ (part of Y^r_i)		
6		$i^2 * \delta^c_i$							$C * D$ (part of Y^c_i)	
7			$i * \delta^r_i * Q^r_i$		$\pi^c_i * \pi^c_i$ (part of Y^c_i)					$\pi^c_i * C$ (part of Y^c_i)
8						1/6 * $(A*B)$ (part of Y^r_i)				
9	$i^3 * \delta^r_i$						$C * C$ (part of Z^c_i)			
10		$i^3 * \delta^c_i$						$i^2 * \delta^c_i * Q^c_i$		
11			$i^2 * \delta^r_i * Q^r_i$						1/6 * $(C*D)$ (part of Y^c_i)	
12										$\pi^c_i * E$ (part of Z^c_i)
13					$\pi^c_i * (\pi^c_i)^2$ (part of Z^c_i)					
14						$i^2 * \delta^r_i * Y^r_i$				
15			$i^3 * \delta^r_i * Q^r_i$				$\delta^c_i * C^2$ (part of Z^c_i)			
16				$i^3 * \delta^c_i * Q^c_i$						
17									$\pi^c_i * F$ (part of Z^c_i)	
18										3 * $(\pi^c_i * E)$ (part of Z^c_i)
19					$i^3 * \delta^c_i * Y^c_i$					
20				$i^3 * \delta^r_i * Y^r_i$						
21										
22										
23										
24										$i^3 * \delta^c_i * Z^c_i$
25										
26										
27										
28										
29										

Table 3.10 Schedule for module with two KCMs and one general multiplier with pipelining

	KCM			General Multiplier						KCM	
1	$i * \delta^r_i$			$\delta^r_i * Q^r_i$						$\pi^c_i * \pi^c_i$ (part of Y^c_i)	
2		$i * \delta^c_i$			$C * D$ (part of Y^c_i)						$\pi^c_i * C$ (part of Y^c_i)
3						$C * C$ (part of Z^c_i)					
4							$A * B$ (part of Y^r_i)				
5	$i^2 * \delta^r_i$							$\pi^r_i * \pi^r_i$ (part of Y^r_i)		$\pi^c_i * \pi^{c\,2}_i$ (part of Z^c_i)	
6		$i^2 * \delta^c_i$							$\pi^r_i * A$ (part of Y^r_i)		$\pi^c_i * E$ (part of Z^c_i)
7			$i * \delta^r_i * Q^r_i$								
8					$1/6 * (C*D)$ (part of Y^c_i)						
9	$i^3 * \delta^r_i$					$\delta^c_i * C^2$ (part of Z^c_i)					
10		$i^3 * \delta^c_i$					$1/6 * (A*B)$ (part of Y^r_i)				
11			$i^2 * \delta^r_i * Q^r_i$					$i^2 * \delta^c_i * Q^c_i$			
12									$3 * (\pi^c_i * E)$ (part of Z^c_i)		
13											
14					$\pi^c_i * F$ (part of Z^c_i)						
15			$i^3 * \delta^r_i * Q^r_i$			$i^3 * \delta^c_i * Y^c_i$					
16							$i^2 * \delta^r_i * Y^r_i$				
17	$i^3 * \delta^c_i * Q^c_i$										
18											
19											
20					$i^3 * \delta^c_i * Z^c_i$						
21											
22											
23											
24											
25											

Table 3.11 Schedule for module with one KCM and two general multipliers with pipelining

	KCM			General Multiplier				General Multiplier				
1	$i * \delta^r_i$			$\delta^r_i * Q^r_i$				$A * B$ (part of Y^r_i)				
2		$i * \delta^c_i$			$C * D$ (part of Y^c_i)				$\pi^r_i * A$ (part of Y^r_i)			
3						$\pi^r_i * \pi^r_i$ (part of Y^r_i)				$\pi^c_i * C$ (part of Y^c_i)		
4							$C * C$ (part of Z^c_i)				$\pi^c_i * \pi^c_i$ (part of Y^c_i)	
5	$i^2 * \delta^r_i$											
6		$i^2 * \delta^c_i$										
7			$i * \delta^r_i * Q^r_i$					1/6 * $(A * B)$ (part of Y^r_i)				
8					1/6 * $(C * D)$ (part of Y^c_i)							
9	$i^3 * \delta^r_i$					$\pi^c_i * E$ (part of Z^c_i)						
10		$i^3 * \delta^c_i$					$\delta^c_i * C^2$ (part of Z^c_i)				$\pi^c_i * (\pi^c_i)^2$ (part of Z^c_i)	
11			$i^2 * \delta^r_i * Q^r_i$									$i^2 * \delta^c_i * Q^c_i$
12												
13								$i^2 * \delta^r_i * Y^r_i$				
14					$i^3 * \delta^c_i * Z^c_i$				$\pi^c_i * F$ (part of Z^c_i)			
15			$i^3 * \delta^r_i * Q^r_i$			$3 * (\pi^c_i * E)$ (part of Z^c_i)						
16												
17	$i^3 * \delta^c_i * Q^c_i$											
18												
19			$i^3 * \delta^r_i * Y^r_i$									
20												
21										$i^3 * \delta^c_i * Z^c_i$		
22												
23												
24												
25												
26												

Table 3.12 Schedule for module with two KCMs and two general multipliers with pipelining

	KCM				General Multiplier				General Multiplier			KCM	
1	$i * \delta^r_i$				$\delta^r_i * Q^r_i$				$A * B$ (part of Y^r_i)			$\pi^c_i * \pi^c_i$ (part of Y^c_i)	
2		$i * \delta^c_i$				$C * D$ (part of Y^c_i)				$\pi^r_i * \pi^r_i$ (part of Y^r_i)			$\pi^c_i * C$ (part of Y^c_i)
3							$C * C$ (part of Z^c_i)				$\pi^r_i * A$ (part of Y^r_i)		
4													
5	$i^2 * \delta^r_i$											$\pi^c_i * \pi^{c\,2}_i$ (part of Z^c_i)	
6		$i^2 * \delta^c_i$											$\pi^c_i * E$ (part of Z^c_i)
7			$i * \delta^r_i * Q^r_i$						1/6 *				
8						1/6 *							
9	$i^3 * \delta^r_i$						$\delta^c_i * C^2$ (part of Z^c_i)		$(A * B)$ (part of Y^r_i)		3 *		
10		$i^3 * \delta^c_i$						$i^2 * \delta^c_i * Q^c_i$					
11			$i^2 * \delta^r_i * Q^r_i$			$(C * D)$ (part of Y^c_i)					$(\pi^c_i * E)$ (part of Z^c_i)		
12													
13									$i^2 * \delta^r_i * Y^r_i$				
14						$\pi^c_i * F$ (part of Z^c_i)				$i^3 * \delta^c_i * Y^c_i$			
15			$i^3 * \delta^r_i * Q^r_i$										
16				$i^3 * \delta^c_i * Q^c_i$									
17													
18													
19			$i^3 * \delta^r_i * Y^r_i$										
20						$i^3 * \delta^c_i * Z^c_i$								
21													
22													
23													
24													
25													

Table 3.13 below compares the number of clock cycles needed to produce the outputs for computing moments of an image.

Table 3.13 Total clock cycles with pipelining

	# of clock cycles needed for each iteration	# of iterations	Total # of clock cycles for all iterations
1 KCM and 1 general multiplier	max{23+16,29} = 39	16	16*39 = 624
1 KCM and 2 general multipliers	max{22+16,26} = 38	29	29*38 = 1102
2 KCMs and 1 general multiplier	max{25+16,25} = 41	20	20*41 = 820
2 KCMs and 2 general multipliers	max{22+16,25} = 38	32	32*38 = 1216

From Table 3.13, the configuration with just one constant coefficient multiplier and one general multiplier is the best choice with only 624 clock cycles, similar to the non-pipelined case. Table 3.14 shows a comparison between non-pipelined and pipelined versions of the one KCM and one general multiplier configuration.

Table 3.14 Comparison between pipelined and non-pipelined

	Number of CLBs per module	Total number of modules	Total CLBs	# of clock cycles needed for each iteration	Total # of clock cycles for all iterations
Non-pipelined	500	16	8000	38	608
Pipelined	506	16	8096	39	624

From Table 3.14, both cases require almost the same number of CLBs and total number of clock cycles. Furthermore, a pipelined system has a much higher clock frequency than a non-pipelined system. For example, from a Xilinx application note

[CCM], a pipelined KCM producing 24-bit outputs runs at 148 Mhz (6.8 ns) and a non-pipelined KCM producing 24-bit outputs runs at 71 Mhz (14 ns). Thus, a pipelined system produces results at a faster rate than a non-pipelined system. This is a good area/time tradeoff considering the fact that computation time will be a lot shorter at a cost of only a slight increase in area. Thus, in this design, we are considering one pipelined KCM and one pipelined general multiplier.

3.2.2 Logic Units in a Single Module

Up to this moment, we have considered only KCMs and general multipliers inside a module. Other logic units are needed, however, such as registers to store existing outputs that are to be used at a later time, as well as multiplexers and adders. In this section, we are looking into computing the terms including $Q^r{}_i$, $Q^c{}_i$, $Y^r{}_i$, $Y^c{}_i$, and $Z^c{}_i$ as discussed in Chapter 2. Note that $\delta^r{}_i$, $\delta^c{}_i$, $\pi^r{}_i$, and $\pi^c{}_i$ are pre-computed values (computed outside of the FPGA chip). Thus, four separate registers store the δ and π values.

An operation that multiplies by 2 shifts the value one bit to the left, and a division by 2 shifts one bit to the right. Figure 3.4 shows the logic needed to compute $Q^r{}_i$ and $Q^c{}_i$. The shifter shifts one bit to the right since it involves a division by 2. Registers also store ($\delta^r{}_i$ – 1) and ($\delta^c{}_i$ – 1) as temporary variables to be used in later computations, specifically as inputs to multiplexers feeding the general multipliers. Figure 3.5 shows how to obtain (2*$\delta^r{}_i$ – 1) and (2*$\delta^c{}_i$ – 1). Note that in both Figures 3.4 and 3.5, the adder is a decrementer that decreases the input by 1.

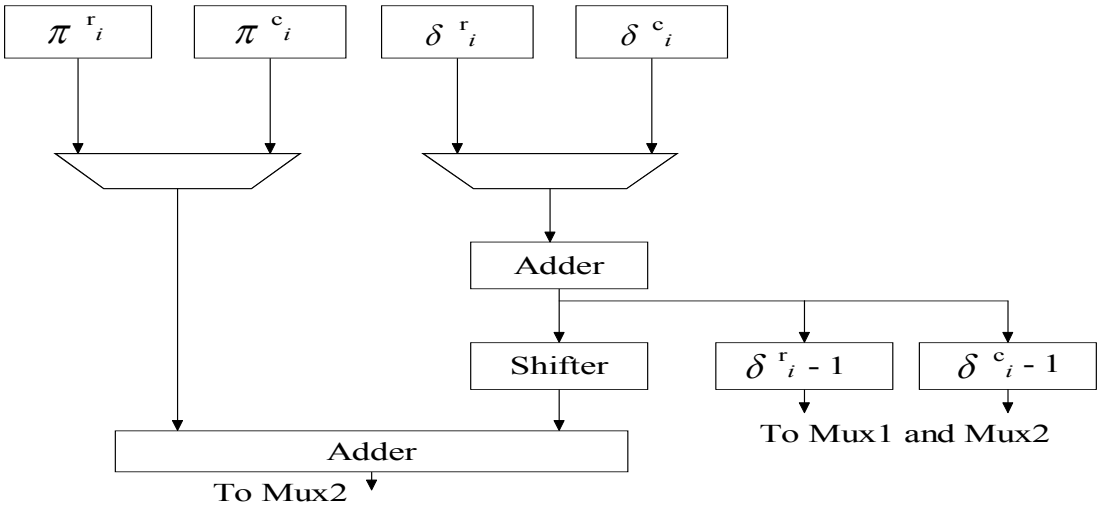

Figure 3.4 Logic units to compute $Q^{r}{}_{i}$ and $Q^{c}{}_{i}$

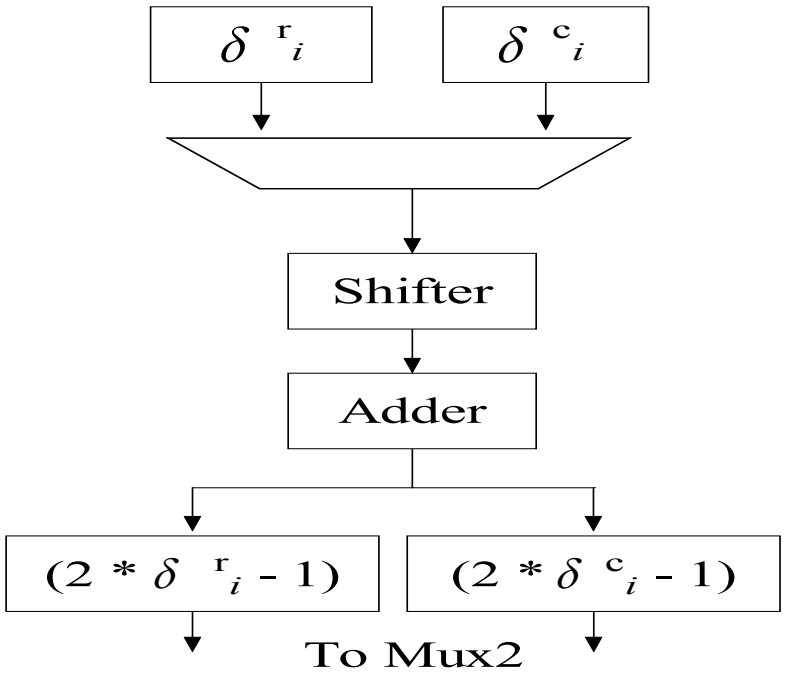

Figure 3.5 Logic units to compute $(2*\delta^{r}{}_{i} - 1)$ and $(2*\delta^{c}{}_{i} - 1)$

As shown in Table 3.9, the two main components of a module are a general multiplier and a constant coefficient multiplier. At this moment we will look at the computations in the constant coefficient multiplier, as shown in Figure 3.6. From Table 3.9, the four inputs into a 4-to-1 multiplexer are $\delta^r{}_i$, $\delta^c{}_i$, the output from the KCM feeding back into itself, and the output from the general multiplier. For example, the constant coefficient is multiplied with $\delta^r{}_i$ and $\delta^c{}_i$ at times $t = 1$ and $t = 2$, respectively. At time $t = 5$, the output produced by the KCM, $i * \delta^r{}_i$, is fed back into itself for a second multiplication and for a third time at $t = 9$. At time $t = 7$, the multiplication is between the constant coefficient and the output produced by the general multiplier, $\delta^r{}_i * Q^r{}_i$.

Some of the outputs produced by the KCM will be used as inputs to the general multiplier. At the end of $t = 9$, the KCM produces the term $i^2 * \delta^c{}_i$, which will be used by the general multiplier at $t = 10$. Since the output from the KCM will be used by the general multiplier at the next clock cycle, it is not necessary to store it in a register as a temporary variable. The terms $i^2 * \delta^r{}_i$ and $i^3 * \delta^c{}_i$, however, will only be used by the general multiplier at $t = 14$ and $t = 24$, respectively. Thus, those two terms will be stored in two separate registers.

Figure 3.7 shows the logic units involved with the general multiplier and primarily used in computing $Y^r{}_i$, $Y^c{}_i$, and $Z^c{}_i$, as indicated in Equations 2.17, 2.18, and 2.19. The four multiplexers, Mux1, Mux2, Mux3, and Mux4, have different inputs and from the scheduling in Table 3.9, outputs from Mux3 and Mux4 are inputs to the general multiplier. The first clock cycle involves the multiplication between $\delta^r{}_i$ and $Q^r{}_i$ from Figure 3.4. $Y^r{}_i$ uses four multiplications as shown at $t = 2$, $t = 3$, $t = 4$, and $t = 8$. At $t = 2$,

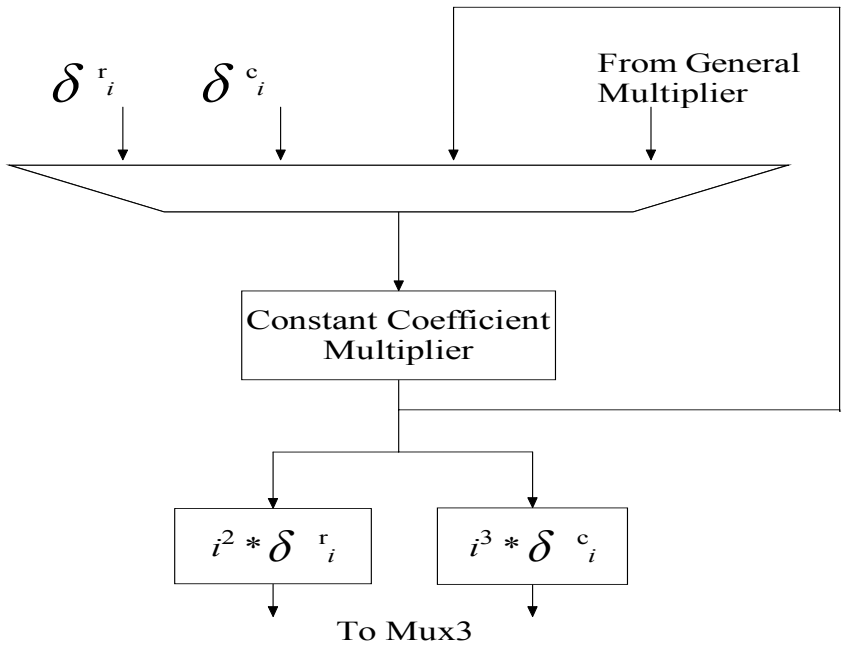

Figure 3.6 Logic units centered on constant coefficient multiplier

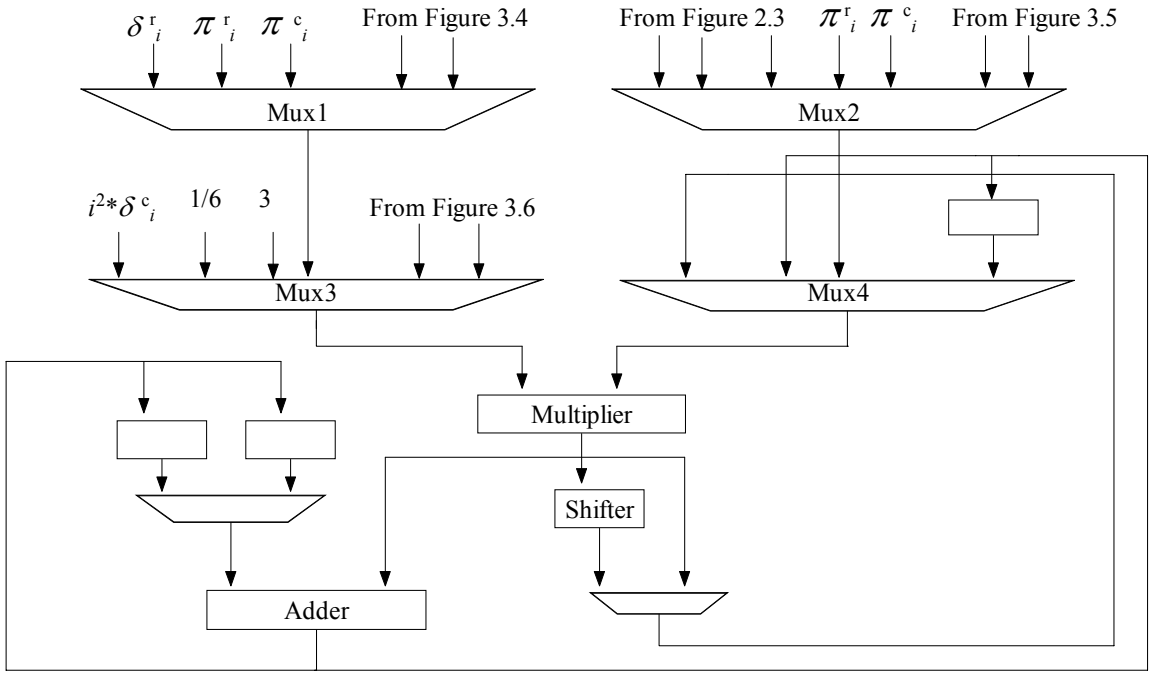

Figure 3.7 Logic units interacting with general multiplier

we are multiplying the terms $\left(\delta^r{}_i - 1\right)$ and $\left(2 * \delta^r{}_i - 1\right)$, and the output is multiplied with $\frac{1}{6}$ at t = 8. The third and forth clock cycles show the multiplication between $\pi^r{}_i$, $\pi^r{}_i$ and $\pi^r{}_i$, $\left(\delta^r{}_i - 1\right)$, respectively. After each multiplication, the output travels to an adder to produce the final value of $Y^r{}_i$. In the event that the input to the adder is not needed until later clock cycles, a register will store the input, as shown in Figure 3.7. Similarly, $Y^c{}_i$ also uses three multiplications, where the terms involved are $\left(\delta^c{}_i - 1\right)$, $\left(2 * \delta^c{}_i - 1\right)$ and $\pi^c{}_i$, $\left(\delta^c{}_i - 1\right)$ and $\pi^c{}_i$, $\pi^c{}_i$ at t = 5, t = 6, and t = 7, respectively. Clock cycle t = 11 shows the same dependency as in $Y^r{}_i$ where the output from the previous multiplication is multiplied with $\frac{1}{6}$.

The last variable, $Z^c{}_i$, uses six multiplications and five of them depend on the outputs produced by previous computations, namely at t = 12 , t = 13, t = 15, t = 17, and t = 18. At t = 10, the multiplication involves the term $\left(i^2 * \delta^c{}_i\right)$ from the constant coefficient multiplier and $Q^c{}_i$. Table 3.9 also shows that at t = 14 and t = 24, newly computed $Y^r{}_i$ and $Z^c{}_i$ are multiplied with $\left(i^2 * \delta^r{}_i\right)$ and $\left(i^3 * \delta^c{}_i\right)$, respectively, at the next clock cycle, thus eliminating the need to store them in registers. A register stores the term $Y^c{}_i$, however, since it is available at t = 17 but will not be used until t = 19.

Figure 3.8 represents a combination of Figures 3.4, 3.5, 3.6, and 3.7. It clearly shows the interconnections between different logic units. The multiplexer at the bottom of Figure 3.8 indicates four possibilities for the final output of the module, namely $\delta^r{}_i$, immediate and delayed outputs from the constant coefficient multiplier, and the output from the general multiplier.

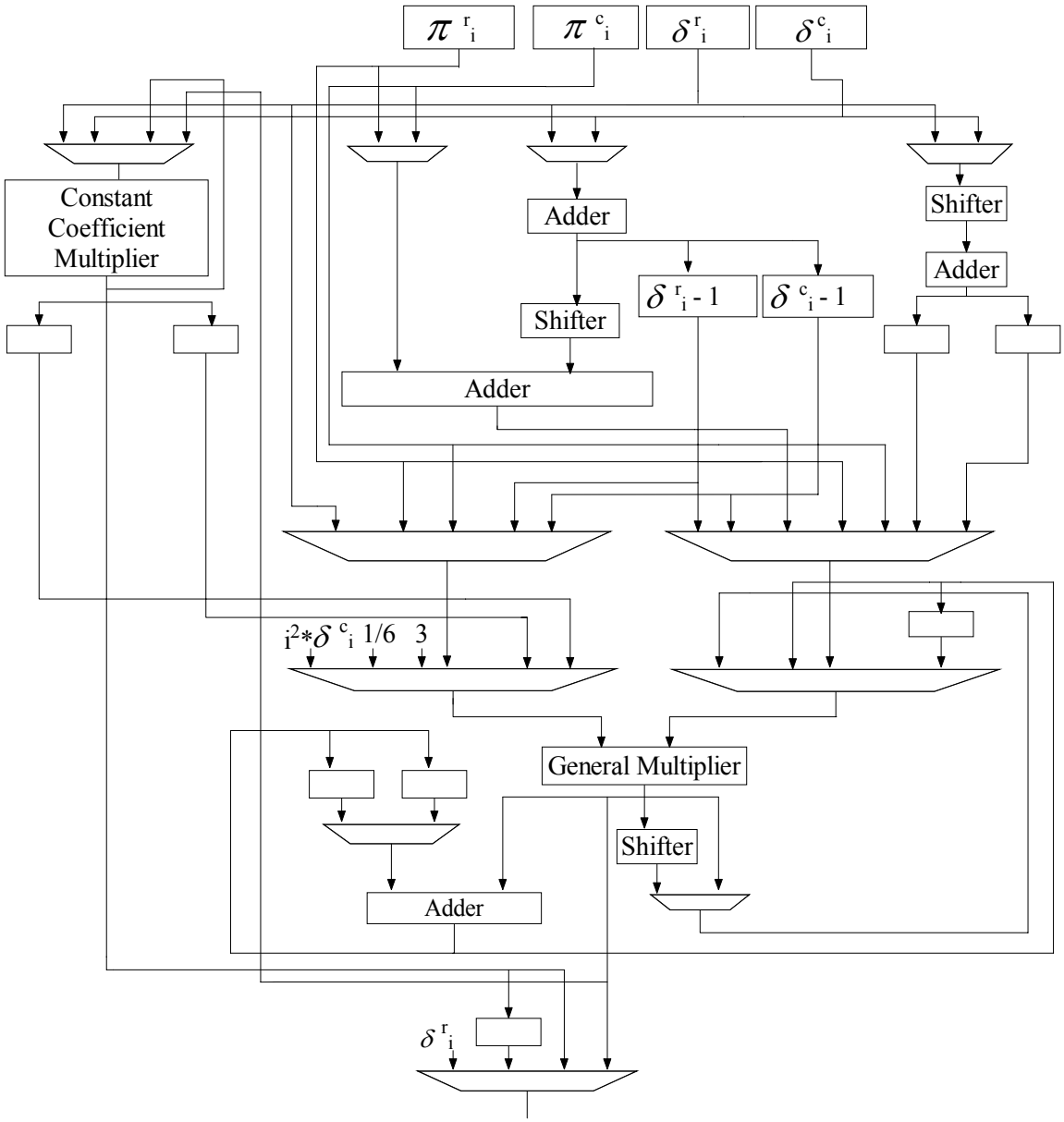

Figure 3.8 Logic units in a single module

3.3 Adding Outputs of All Modules

In the previous section, we talked about all the logic units in a single module. We looked specifically at the case where a single module contains one general multiplier and one constant coefficient multiplier. With 16 modules, the FPGA performs a total of 16

iterations to compute the moments of a 256 × 256 binary HV-convex image. In this section, we will consider the summation of the outputs from each module using carry-save adders.

3.3.1 Overview of Carry-Save Adders and Wallace Tree

In this design, we are looking at utilizing three inputs to two outputs *carry-save adders* (CSA). After a fan-in tree of 3-to-2 units reduces *n* inputs to two outputs (a sum vector and a carry vector), a conventional full adder produces the final output by adding the sum vector and carry vector. For example, assume we have three 6-bit inputs, X = 101101, Y = 011001, and Z = 101011. The sum of those three inputs can be computed as follows:

X:		1	0	1	1	0	1
Y:		0	1	1	0	0	1
Z:		1	0	1	0	1	1
	0	1	1	1	1	1	sum
1	0	1	0	0	1		carry
1	1	1	0	0	0	1	output

In general, for N -bit inputs, where $X[m]$ denotes bit m of X for $0 \le m \le N$,

$S[m] = X[m] \oplus Y[m] \oplus Z[m]$,

$C[m] = X[m] \bullet Y[m] + Y[m] \bullet Z[m] + X[m] \bullet Z[m]$, and

$Output = S + 2 * C = X + Y + Z$.

One main advantage of using carry-save adders is that they eliminate the propagation delay of the carries, as shown in Figure 3.9. Instead of propagating the carry signal to the

next most significant bit, a carry-save adder represents the output as a sum (S) vector and a carry (C) vector.

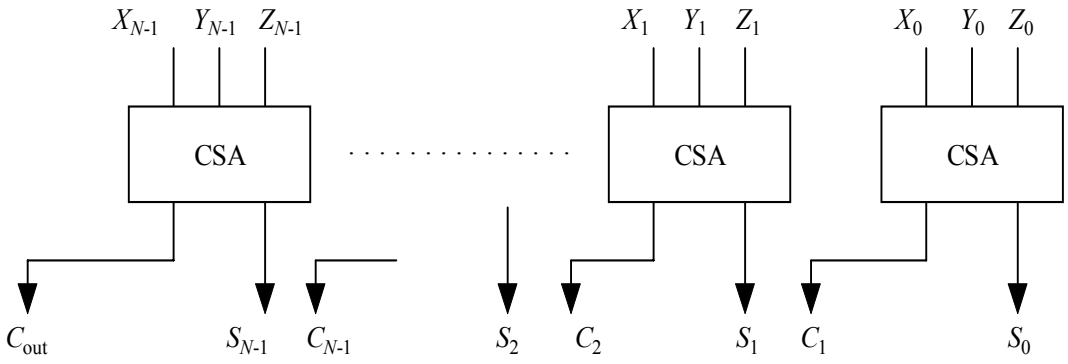

Figure 3.9 N-bit carry-save adder

A Wallace tree consists of multiple levels, where each level consists of carry-save adders [PT]. The number of CSA levels depends on the number of inputs. Figure 3.10 shows an example of a Wallace tree with nine inputs.

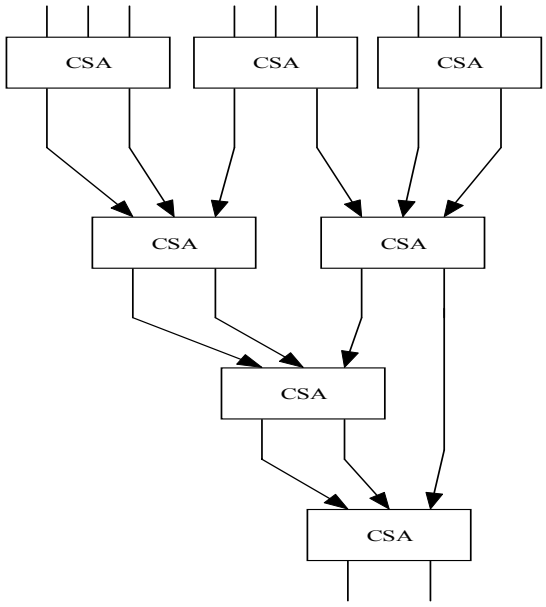

Figure 3.10 Wallace tree with nine inputs

3.3.2 Summation of Outputs from Each Module in a Single Iteration

From Chapter 2, for each iteration, every module will produce outputs that will be sent to the Wallace tree adder for summation. In this design, with 16 modules, the Wallace tree will have 16 inputs at a time. From Table 3.12, the order of outputs that are being produced by each module is as follows (in increasing order of time, *t*):

Stored in register : $\delta^r{}_i$ (for m_{00}) or $\delta^c{}_i$ (for m_{00})

At $t = 5$: $i * \delta^r{}_i$ (for m_{10})

At $t = 6$: $i * \delta^c{}_i$ (for m_{01})

At $t = 9$: $i^2 * \delta^r{}_i$ (for m_{20})

At $t = 10$: $i^2 * \delta^c{}_i$ (for m_{02})

At $t = 11$: $i * \delta^r{}_i * Q^r{}_i$ (Eq 2.8.1 --- for m_{11})

At $t = 13$: $i^3 * \delta^r{}_i$ (for m_{30})

At $t = 14$: $i^3 * \delta^c{}_i$ (for m_{03})

At $t = 15$: $i^2 * \delta^r{}_i * Q^r{}_i$ (Eq 2.9.1 --- for m_{21})

At $t = 16$: $i^2 * \delta^c{}_i * Q^c{}_i$ (Eq 2.11.1 --- for m_{12})

At $t = 19$: $i^3 * \delta^r{}_i * Q^r{}_i$ (Eq 2.10.1 --- for m_{31})

At $t = 20$: $i^2 * \delta^r{}_i * Y^r{}_i$ (Eq 2.13.1 --- for m_{22}) and $i^3 * \delta^c{}_i * Q^c{}_i$ (Eq 2.12.1 --- for m_{13})

At $t = 24$: $i^3 * \delta^r{}_i * Y^r{}_i$ (Eq 2.14.1 --- for m_{32})

At $t = 25$: $i^3 * \delta^c{}_i * Y^c{}_i$ (Eq 2.15.1 --- for m_{23})

At $t = 30$: $i^3 * \delta^c{}_i * Z^c{}_i$ (Eq 2.16.1 --- for m_{33})

From Equation 2.1, we can compute m_{00} by adding either $\delta^r{}_i$ or $\delta^c{}_i$. As shown in Figure 3.8, we have chosen $\delta^r{}_i$. From the order of the outputs, it is evident that the modules produce most outputs at different clock cycles, thus avoiding any conflicts. At *t* = 20, however, two concurrent outputs are ready for addition. The general multiplier and

the constant coefficient multiplier produce the terms $i^2 * \delta^r{}_i * Y^r{}_i$ and $i^3 * \delta^c{}_i * Q^c{}_i$, respectively. In this case, a register must store one output before passing it to the Wallace tree adder. As shown at the bottom of Figure 3.8, a register stores the term $i^3 * \delta^c{}_i * Q^c{}_i$. Figure 3.11 shows a rough sketch of all the components in a single iteration. Figure 3.12 shows in detail the number of carry-save adders and levels inside the Wallace tree. The Wallace tree contains six levels with a total of 14 carry-save adders for an addition with 16 inputs. A conventional full adder will compute the final two outputs. Note that the labeled sizes of inputs going into the Wallace tree is just an upper bound value. The inputs can be less than 64 bits.

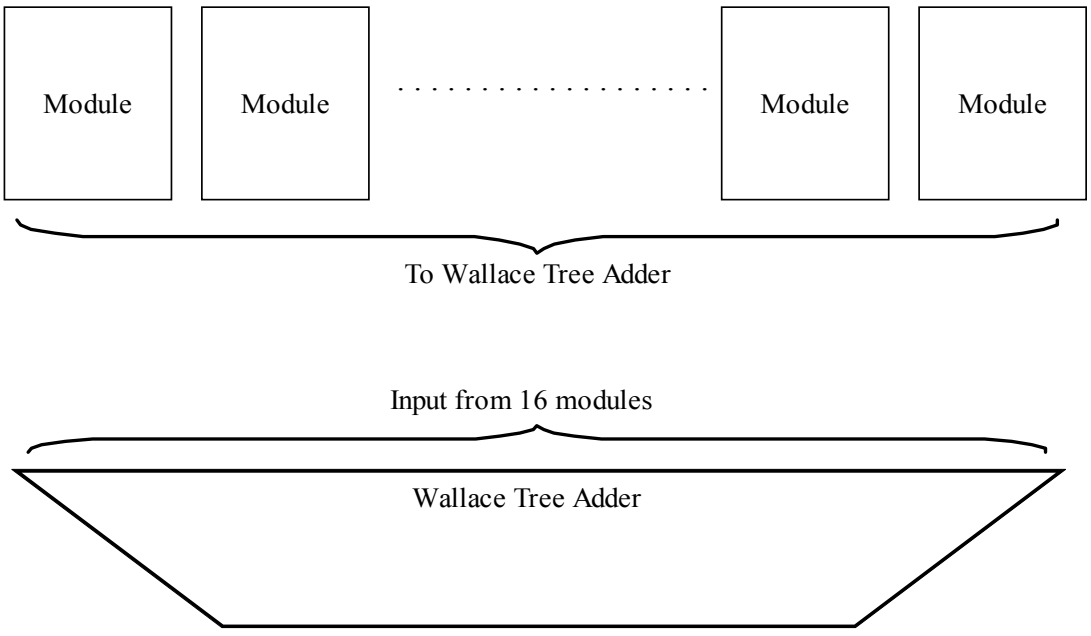

Figure 3.11 Components in a single iteration

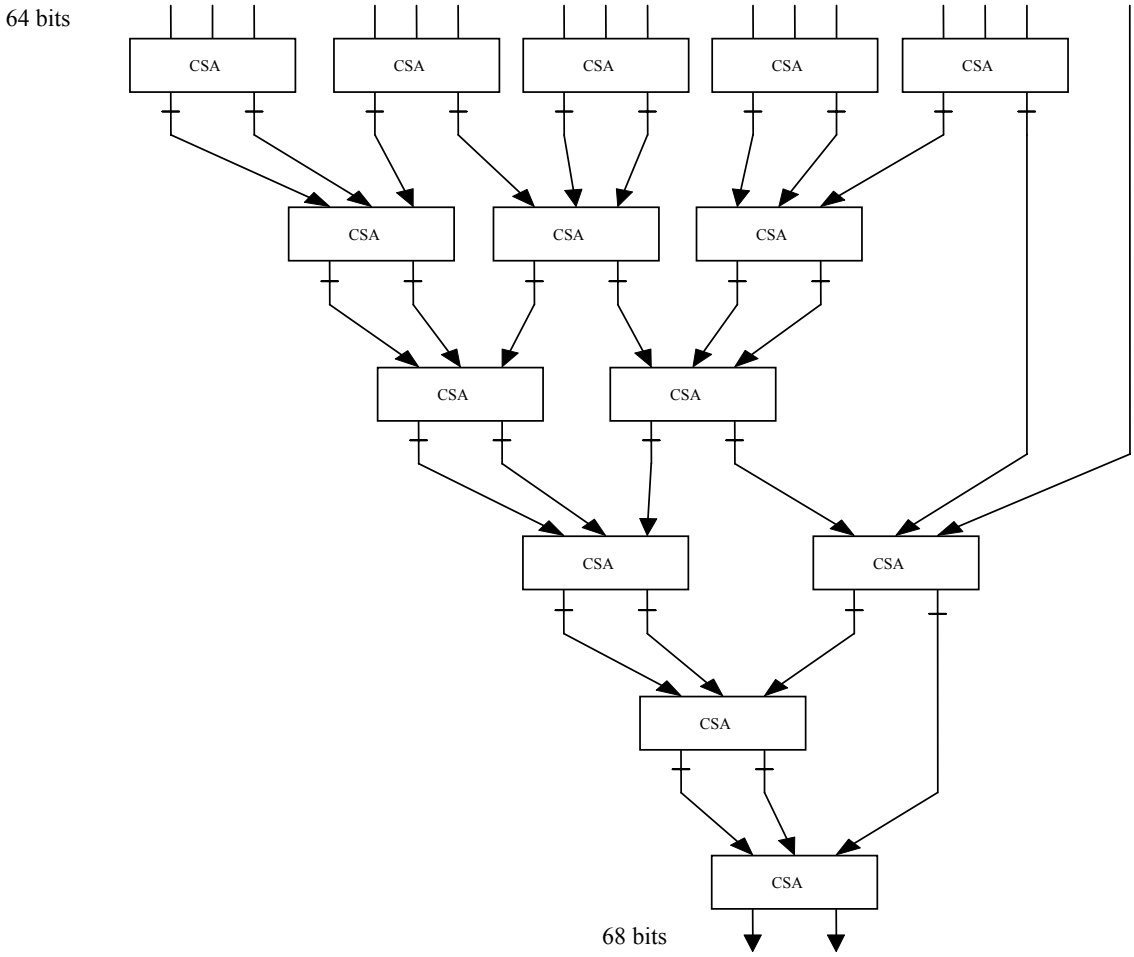

Figure 3.12 Wallace Tree with 16 inputs

3.3.3 Summation across Iterations

From Section 3.2, we know that it takes 16 iterations to completely process the whole 256 × 256 image. From Section 3.3.2, in each iteration, the Wallace tree will produce 16 outputs. Thus, summing up these values across all 16 iterations will compute the moments. Since we are using run-time reconfiguration where computations are done iteration by iteration, the FPGA must store the sum of outputs computed in the current

iteration before feeding it back into the Wallace tree for addition with outputs from the next iteration.

Figures 3.13 and 3.14 show a slight modification of both Figures 3.11 and 3.12. In Figure 3.13, a memory block temporarily stores outputs from the Wallace tree before they are added with outputs from the next iteration. Figure 3.14 shows a Wallace tree with 17 inputs where the additional input comes from the memory block. It also has six levels with a total of 15 carry-save adders. The largest 17^{th} input is 72 bits long ($64+\lceil \log_2(256-16) \rceil$), which occurs in the second to last iteration of all 16 iterations. The largest output obtained by summing up the values across all 16 iterations is 72 bits long ($64+\lceil \log_2(256) \rceil$).

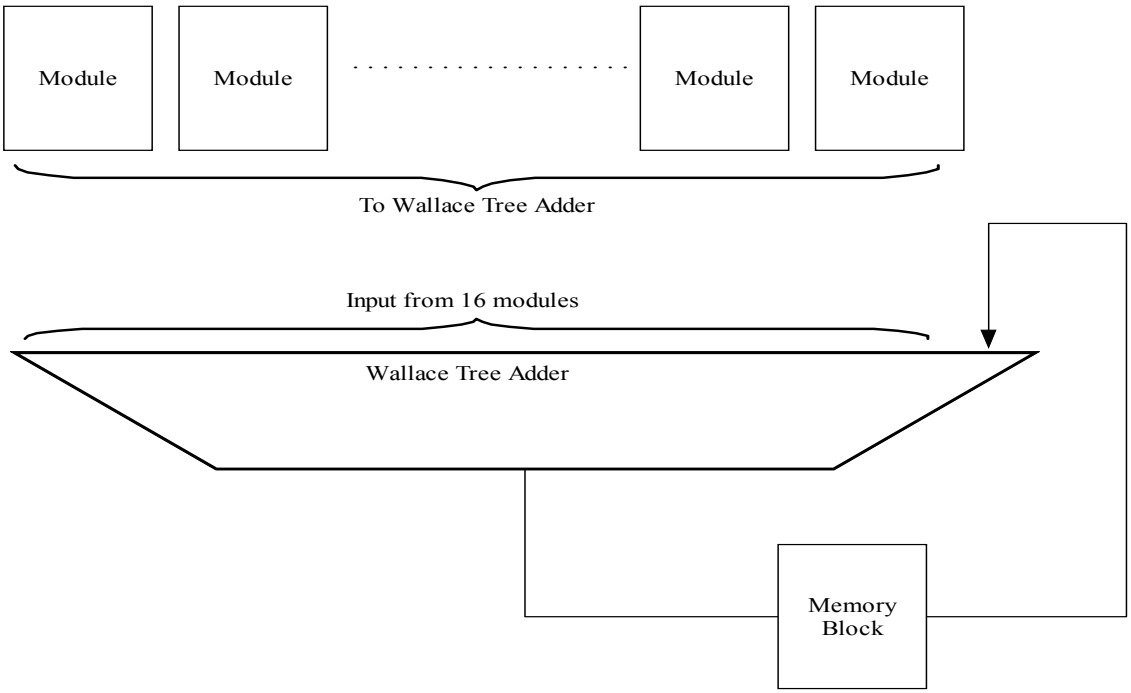

Figure 3.13 Components needed for computations across iterations

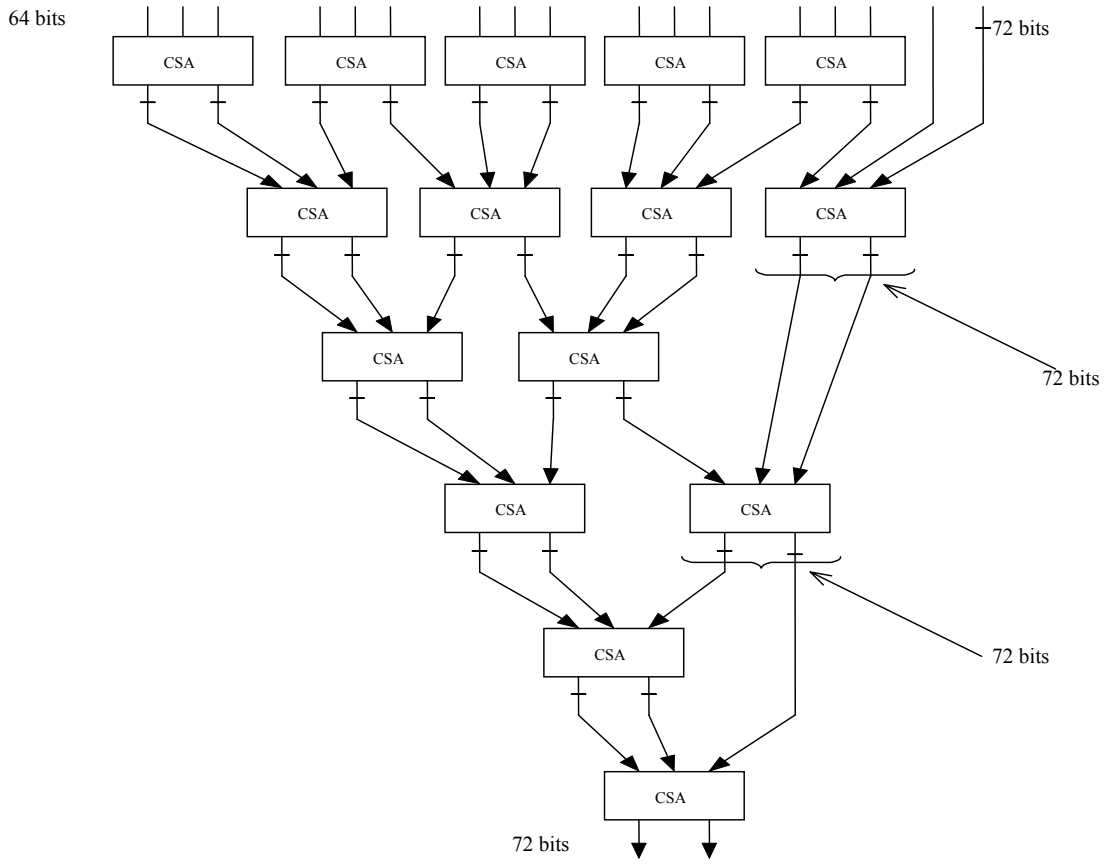

Figure 3.14 Wallace Tree with 17 inputs

In the Virtex-E series FPGA, blocks of SelectRAM memory serve as temporary storage for the outputs. As shown in Figure 3.12, the largest output from the Wallace tree consists of 72 bits. From the Xilinx datasheet [FPGA], the SelectRAM memory can support 256 memory locations, each with 16 bits. Storing a 72-bit output will require five memory locations. Thus, to temporarily store all sixteen 72-bit outputs produced from each iteration, we will only need 80 memory locations. Note that the total number of memory locations is just an upper bound value, since not every output out of the 16 outputs will be 72 bits.

3.4 Overall Size

So far, we have looked at all the logic units utilized for moment computations of a binary HV-convex image. In this section, we provide an estimation of the total area occupied in terms of configuration logic blocks (CLBs) in an FPGA chip. Our design consists of 16 modules and a Wallace tree, along with other logic units such as adders, multiplexers, registers, and shifters. A shifter does not require any additional CLBs since the output can be obtained by changing the position of the bits accordingly. From Enzler *et al.* [EJCT], an estimated value of CLBs for each component is as follows:

Adder = (Average number of bits) / 2

*Multiplexer = (Average fan-in) * (Average number of bits) / 4*

Register = (Average number of bits) / 2

The two major components in each module are a general multiplier and a constant coefficient multiplier. Table 3.15 summarizes the number of CLBs in a pipelined general multiplier and a pipelined KCM as discussed in Section 3.2.1.2.

Table 3.15 CLBs occupied in general multipliers and constant coefficient multipliers

	CLBs in one module	CLBs in 16 modules
General Multiplier	382	6112
Constant Coefficient Multiplier	124	1984

Figure 3.15 is similar to Figure 3.8, with the exception that it clearly labels the largest input (in terms of bits) going into each component and labels in blue the number of CLBs consumed by each component. The logic units shown are for a single module. Since the

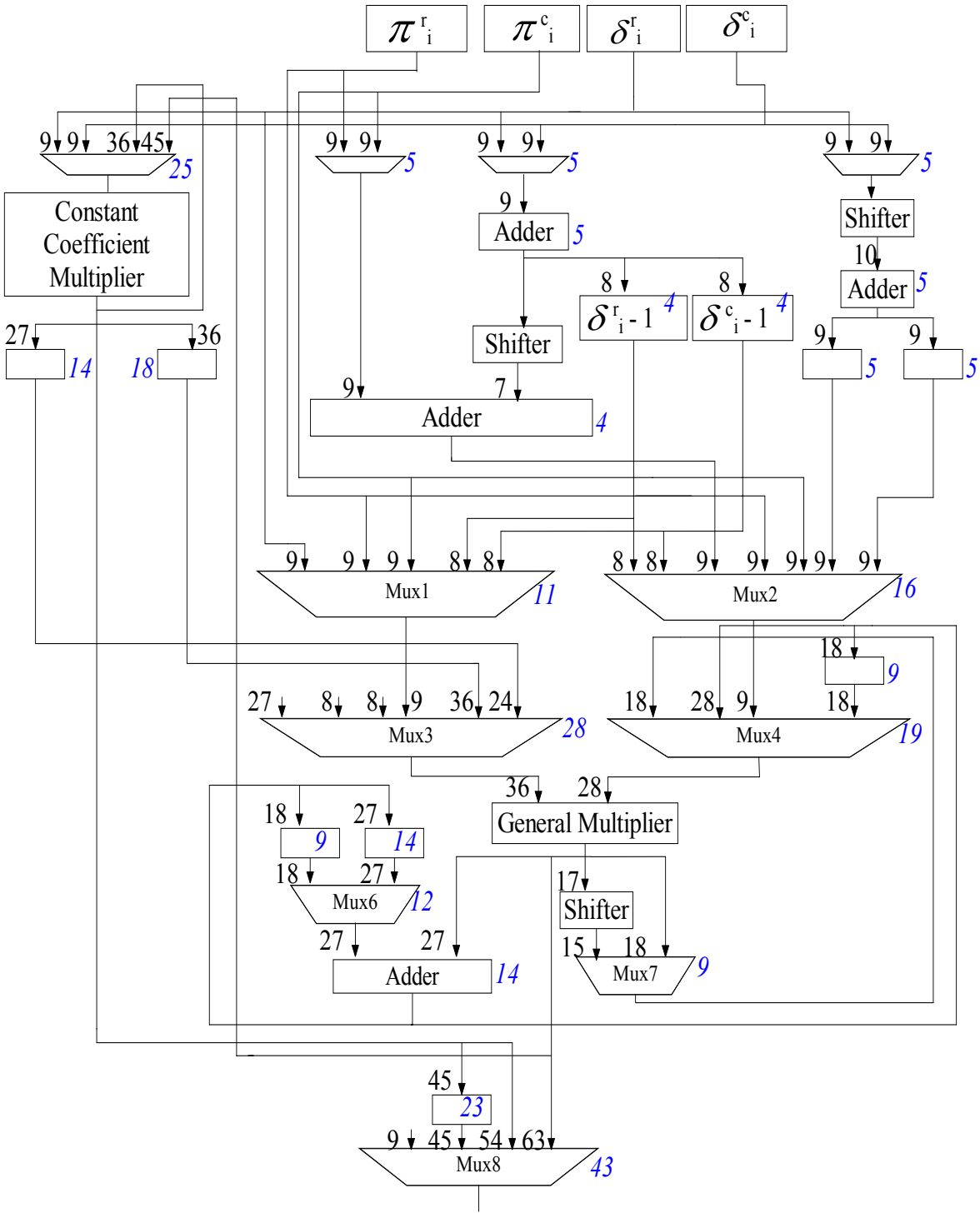

Figure 3.15 Logic units inside a module, blue color indicates number of CLBs

image size we are considering is 256 × 256, the values $\delta^r{}_i$, $\delta^c{}_i$, $\pi^r{}_i$, and $\pi^c{}_i$ will each be 9 bits. Thus, we use a total of 18 CLBs for four registers to store those values. The multiplexer for the KCM has four inputs, nine bits for $\delta^r{}_i$ and $\delta^c{}_i$, 36 bits for the term $i^2 * \delta^r{}_i * Q^r{}_i$ and finally 45 bits for $i^2 * \delta^r{}_i * Y^r{}_i$, giving us a total of 25 CLBs. From Figure 3.6, the registers storing $i^2 * \delta^r{}_i$ (27 bits) and $i^3 * \delta^c{}_i$ (36 bits) will use 14 and 18 CLBs, respectively. From the general multiplier, according to Table 3.9, the largest term going into the adder, shifter, and Mux7 will be 27 bits, 17 bits, and 18 bits, respectively. The terms going into Mux8 are $\delta^r{}_i$ (9 bits), $i^3 * \delta^c{}_i * Q^c{}_i$ (45 bits), $i^3 * \delta^r{}_i * Y^r{}_i$ (54 bits), and $i^3 * \delta^c{}_i * Z^c{}_i$ (64 bits).

In conclusion, the components in a single module (represented as 'Miscellaneous' in Table 3.16), excluding the general multiplier and constant coefficient multiplier, use approximately 329 CLBs or 658 slices. For 16 modules, we will need a total of 5264 CLBs or 10,528 slices. Figure 3.16 shows the number of CLBs (in blue color) occupied by the Wallace tree. In addition to the 500 CLBs (1000 slices) for the Wallace tree, a final full adder for the sum vector (72 bits) and carry vector (72 bits) uses 36 CLBs (72 slices). Table 3.16 summarizes the number of CLBs (slices) used by all 16 modules and the Wallace tree. In this design, we use approximately 14,274 CLBs (25,932 slices) to compute the moments for a 256 × 256 binary HV-convex image. From the Xilinx datasheet [FPGA], the Xilinx Virtex-E, device XCV3200E has 16,224 CLBs (32,448 slices). Furthermore, this Virtex-E chip has 208 SelectRAM blocks, each with 4096 bits, which is more than sufficient for the requirements of memory blocks in this design since we will use only one memory block. Furthermore, $\delta^r{}_i$, $\delta^c{}_i$, $\pi^r{}_i$, and $\pi^c{}_i$ are computed outside of the FPGA chip. Since each of the values is 9 bits, every module needs 36 pins

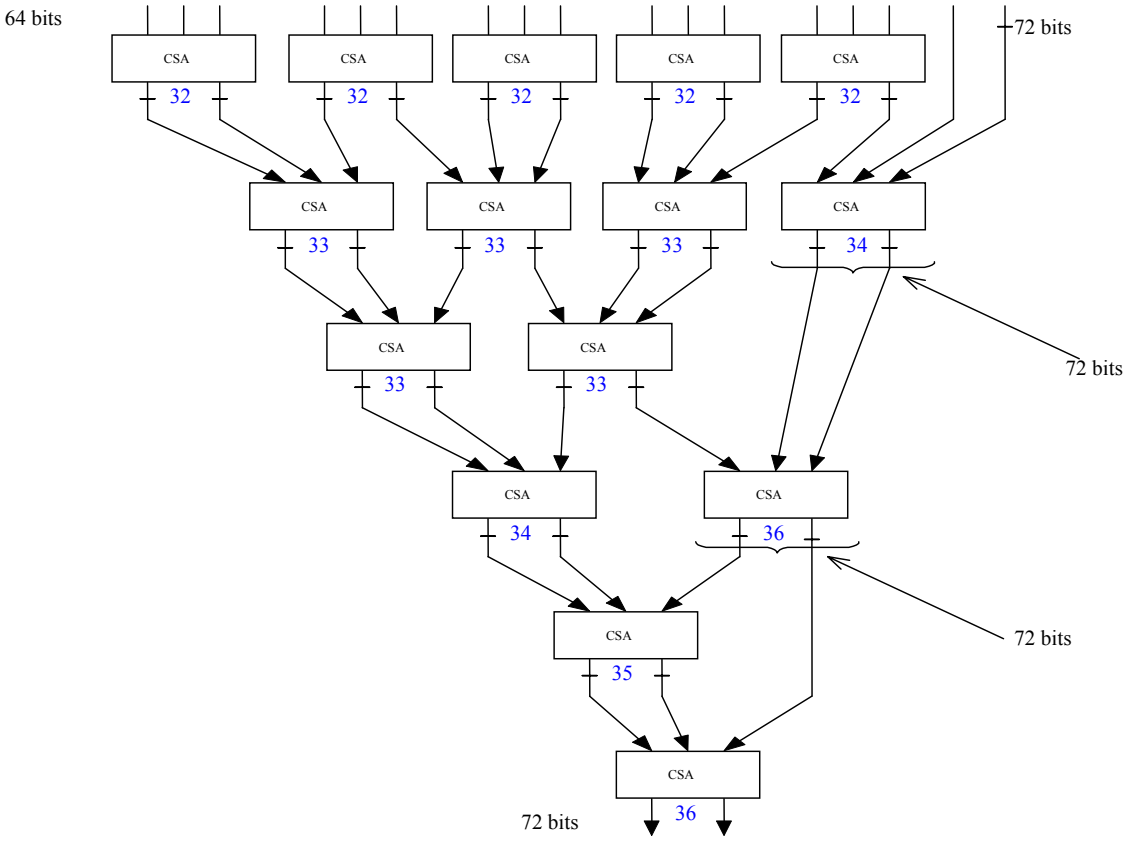

Figure 3.16 CLBs occupied by Wallace tree

Table 3.16 Total number of CLBs (slices) used

	CLBs (slices) in a single module	CLBs (slices) in 16 modules
General multiplier	382 (675)	6112 (10,800)
Constant coefficient multiplier	124 (221)	1984 (3536)
Miscellaneous	329 (658)	5264 (10,528)
Wallace tree		500 (1000)
Final full adder		36 (72)
Total	835 (1554)	13,896 (25,936)

as inputs to the FPGA, giving us a total of 576 pins for all 16 modules. The largest output produced by the Wallace tree is 72 bits. Thus, in this design, we need only 648 (576+72) I/O pins and the XCV3200E has 804 I/O pins. As a result, this device is a suitable platform for the implementation of this design.

Chapter 4

Variations of Binary Horizontally/Vertically Convex Images

In Chapter 3, we presented a design for moment computations of a binary HV-convex image. In this section, we look at variations of a binary HV-convex image, specifically a binary non-convex image and a non-binary image to see the extent to which our design still holds. The most significant features extracted from a binary HV-convex image are the terms $\delta^r{}_i$ ($\delta^c{}_i$) and $\pi^r{}_i$ ($\pi^c{}_i$) which correspond to the length of the string of black pixels in row (column) i and the row (column) coordinate of the leftmost (lowest) black pixel, respectively. By considering a binary non-convex image, however, we look at the limitations in computing the moments using δ's and π's. With binary HV-convex images, moment computations rely heavily on the fact that any black pixels in a row and in a column are contiguous. For example as shown in Equation 2.8.1, computation of m_{11} is as follows:

$$m_{11,i} = \pi^r{}_i * i + (\pi^r{}_i + 1) * i + ... + (\pi^r{}_i + \delta^r{}_i - 1) * i,$$

for $1 \le i \le N$ where $m_{11} = \sum_{i=1}^{N} m_{11,i}$.

The computation begins at the leftmost black pixel, $\pi^r{}_i$, and ends at the rightmost black pixel, $(\pi^r{}_i + \delta^r{}_i - 1)$. This expression is invalid for a non-convex binary image since no constraints exist on the arrangement of black and white pixels. Thus, from the three groups of moments computed on a binary HV-convex image discussed in Chapter 2, only

the Group 1 equations still apply, since they depend only on the number of black pixels in a row or column and not on whether these pixels are contiguous. In particular, the set of moments computed on a non-convex binary image is as follows.

$$m_{00} = \sum_{i=1}^{N} \delta_i^r = \sum_{i=1}^{N} \delta_i^c$$

$$m_{01} = \sum_{i=1}^{N} \delta_i^c * i \qquad m_{10} = \sum_{i=1}^{N} \delta_i^r * i$$

$$m_{02} = \sum_{i=1}^{N} \delta_i^c * i^2 \qquad m_{20} = \sum_{i=1}^{N} \delta_i^r * i^2$$

$$m_{03} = \sum_{i=1}^{N} \delta_i^c * i^3 \qquad m_{30} = \sum_{i=1}^{N} \delta_i^r * i^3$$

A module containing one KCM with i as the constant coefficient can compute the terms in all these seven moment summations. We are looking specifically into a pipelined KCM. Thus, a single module will contain only a constant coefficient multiplier and two multiplexers as shown in Figure 3.1. Table 3.1 presents the scheduling process of all the terms involved, which is similar to a binary HV-convex image. The total number of clock cycles needed is 29 (13 + 16) since a KCM requires 16 clock cycles for reconfiguration. From Figure 3.1, the largest input going into Mux1 from the KCM is 27 bits wide. Thus, Mux1 uses approximately 12 CLBs (24 slices). As for the KCM, the largest output is 36 bits. From a Xilinx application note [CCM], non-pipelined KCMs producing 24-bit and 32-bit outputs use approximately 25 CLBs (10 rows, 3 columns) and 43 CLBs (12 rows, 4 columns). For every 8-bit increment, the number of rows increases by two and the number of columns increases by one. A non-pipelined KCM producing a 40-bit output uses 65 CLBs (14 rows, 5 columns). Thus, a non-pipelined KCM producing a 36-bit

output consumes approximately 54 CLBs or 92 slices (1.7 * 54). Recall from Section 3.2.1.2 that there is a difference of about ten slices between a non-pipelined KCM and a pipelined KCM with three stages. Thus, a pipelined KCM with four stages producing a 36-bit output uses approximately 107 slices (92 + 15) or 63 CLBs. Mux2 uses about 12 CLBs since it has a 9-bit input and a 36-bit input.

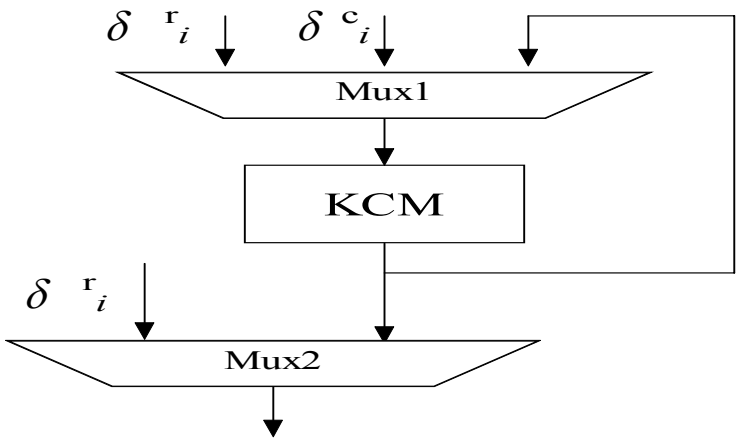

Figure 4.1 Logic units inside a module to compute all seven moments for a binary non-convex image

Table 4.1 Schedule for a module with one pipelined KCM to compute moments of a binary non-convex image

1	$i*\delta^{r}_{i}$	
2		$i * \delta^{c}_{i}$
3		
4		
5	$i^2 * \delta^{r}_{i}$	
6		$i^2 * \delta^{c}_{i}$
7		
8		
9	$i^3 * \delta^{r}_{i}$	
10		$i^3 * \delta^{c}_{i}$
11		
12		
13		

In this discussion, we assume that there are 32 modules to compute the moments. Thus, we need a total of eight iterations to compute the moments of a 256 × 256 binary non-convex image. Figure 3.2 shows a Wallace tree with 32 36-bit inputs. Figure 3.3 shows the summation of the outputs from 32 modules and the summation across iterations and the number of CLBs (in blue color) used by the Wallace tree. The largest 33^{rd} input is 44 bits ($36+\lceil \log_2(256-32) \rceil$), which occurs in the second to last iteration of all eight iterations. The largest output obtained by summing up the values across all eight iterations is 44 bits ($36+\lceil \log_2(256) \rceil$).

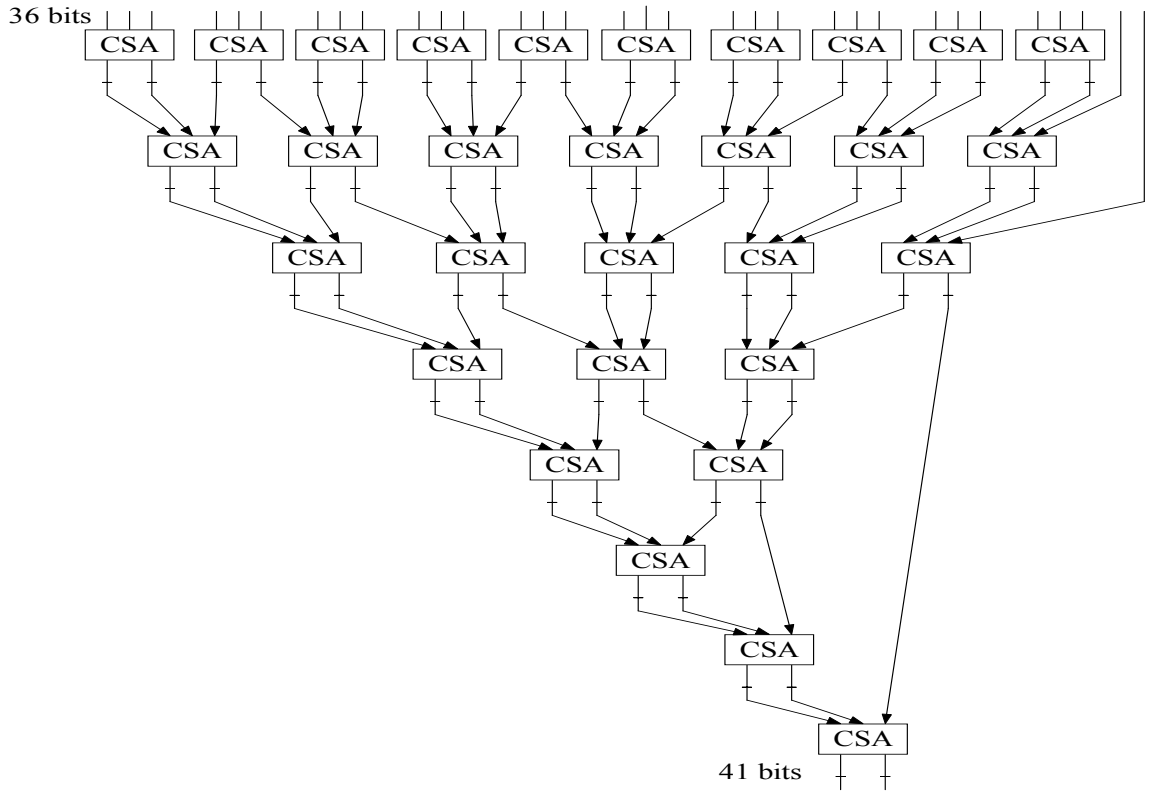

Figure 4.2 Wallace tree with 32 inputs

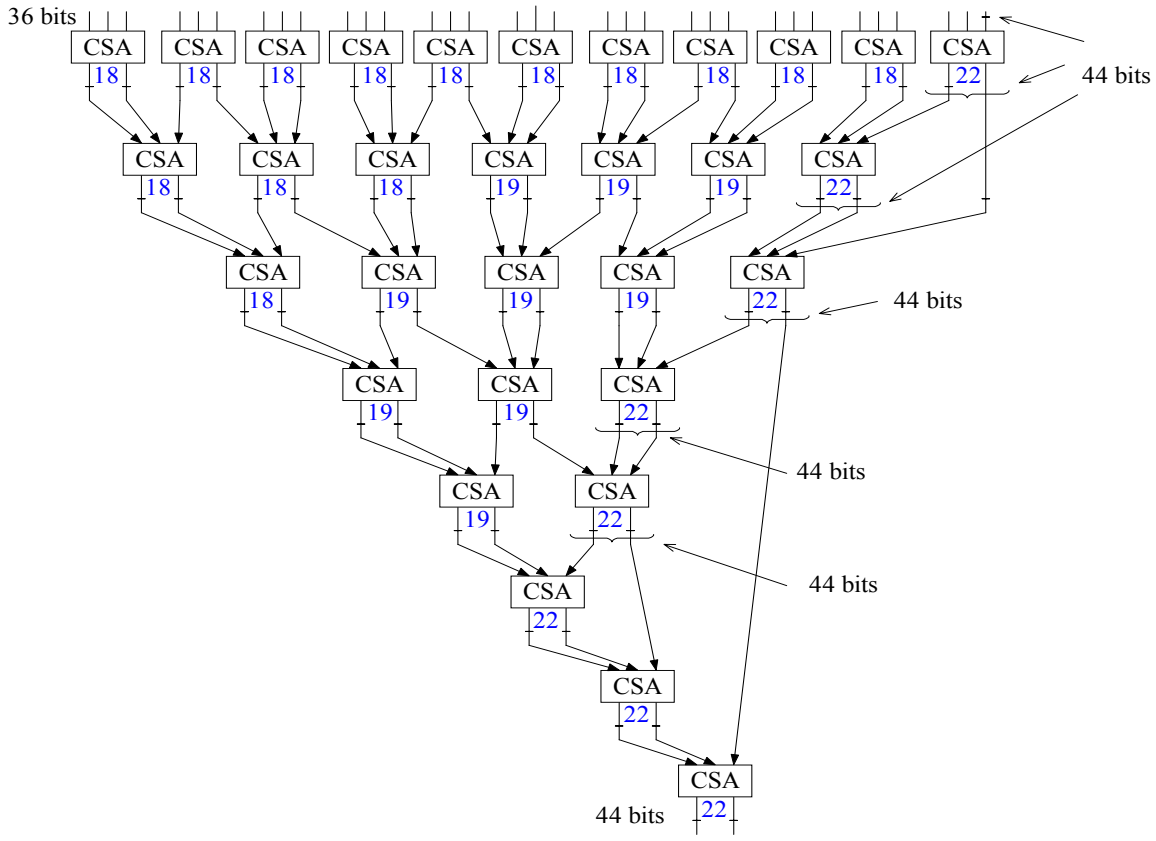

Figure 4.3 Wallace tree with 33 inputs

The other variation we will look at is a non-binary image. In a non-binary image, the pixels consist of different shades of gray, where the value of each pixel ranges from 0 to 255. This design, however, considers an image that has two distinct regions, black (1) and white (0). The computation of δ's and π's depends on this restriction. Thus, this design is not applicable to a non-binary image. Furthermore, a non-binary image cannot be convex since one cannot clearly distinguish between two regions.

So far, we have assumed that $\delta^r{}_i$ and $\delta^c{}_i$ are pre-computed values. As an alternative solution, an FPGA chip computes these values by having two counters to

count all the black pixels of the binary HV-convex image. For example, we scan a 256 × 256 image in row-major order. One of the counters computes the number of black pixels in a row, $\delta^r{}_i$. As for computing length of black pixels in a column, SelectRAM memory stores each $\delta^c{}_i$. Since it is a 256 × 256 image, we will need 256 memory locations to store all 256 values of $\delta^c{}_i$. When a black pixel is scanned in column i, the second counter updates the value $\delta^c{}_i$.

Chapter 5

Summary

Moments have been a very useful tool in digital image processing, especially in the area of pattern recognition due to their invariant properties. Various applications that utilize moments include aircraft identification, pavement distress classification, and breast cancer detection [COC, CWH, DBM]. With increasing order, however, moments are susceptible to noise. Fortunately, most applications will only need moments of up to third order. Moment calculations are usually computation intensive due to the large number of multiplication and addition operations. On images with large sizes, moment computations can certainly become the bottleneck of the entire system. To overcome this problem and to speed up moment computation, we calculate moments on an FPGA chip using run-time reconfiguration. By utilizing the concept of run-time reconfiguration, we divide a large image into segments, and with a set of hardware that gives us the best area/time performance, we process the image one segment at a time. The hardware will reconfigure itself before processing the next segment.

In this thesis, we looked at a binary horizontally/vertically convex image that possesses features that can take advantage of run-time reconfiguration. Using these features, we compute the moments without using the conventional formula, $m_{pq} = \sum_{i=1}^{N}\sum_{j=1}^{N} i^p j^q B(i,j)$. Instead we use the terms $\delta^r{}_i$ ($\delta^c{}_i$) and $\pi^r{}_i$ ($\pi^c{}_i$) which correspond to length of black pixels in row (column) i and the row (column) coordinate of leftmost (lowest) black pixel, respectively. We looked at different combinations of

non-pipelined and pipelined versions of constant coefficient multipliers and general multipliers. The best combination for this design is a pipelined version of one KCM and one general multiplier. We also looked at the advantage of carry-save adders and the use of Wallace tree to add outputs from all 16 modules. We conclude by giving an approximation of the total number of configurable logic blocks (CLBs) used to implement the design. Furthermore, we also consider variations of a binary HV-convex image by looking into a binary non-convex image and a non-binary image. In a binary non-convex image, with the existing hardware used to compute moments for a binary HV-convex image, we can only compute seven of the sixteen moments on a binary non-convex image.

As a proposal to extend the work that has been done for this thesis, we can look at computing moments of order higher than three since some applications do use higher order moments [CWH]. We can also consider a quantitative analysis on how this design benefits the applications that use moments of up to third order.

Bibliography

[BPRS] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, "The Power of Reconfiguration," *J. Par. And Distr, Comput.*, Vol. 13, 1991, pp. 139-153.

[BR] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers*, pp. 42-56.

[CCM] Xilinx, Inc. (1999), "Constant (k) Coefficient Multiplier Generator for Virtex," application note.

[Chung] K. L. Chung, "Computing Horizontal/Vertical Convex Shapes Moments on Reconfigurable Meshes," *Pattern Recognition*, Vol. 20, No. 10, 1996, pp. 1713-1717.

[COC] J. Chou, W. A. O'Neill, and H. D. Cheng, "Pavement Distress Classification Using Neural Networks," *Proc. IEEE Int. Conf. Syst., Man, Cybern*., Vol. 1, 1994, pp. 397-401.

[CWH] H. D. Cheng, C. Y. Wu, D. L. Hung, "VLSI for Moment Computation and Its Applications to Breast Cancer Detection," *Pattern Recognition*, Vol. 31, No. 9, 1998, pp. 1391-1406.

[DBM] S. A. Dudani, K. J. Breeding and R. B. McGhee, "Aircraft Identification by Moment Invariants," *IEEE Transactions on Computers*, Vol. C-26, No. 1, January 1977, pp. 39-45.

[DCCM] Xilinx, Inc. (2000), "Dynamic Constant Coefficient Multiplier V2.0," product specification.

[EH] J. G. Eldredge and B. L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration," *Proc. IEEE Workshop FPGA's Custom Computing Machines*, April 1994, pp. 180-188.

[EJCT] R. Enzler, T. Jeger, D. Cottet, and G. Troster, "High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs," *Field Programmable Logic* 2000, R.W Hartenstein and H.Grunbacher (Eds.), LNCS 1896, pp. 525-534.

[Fausett] L. Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*, 1994, pp. 289-300.

[FPGA] Xilinx, Inc. (2001), "Virtex-E 1.8V Field Programmable Gate Arrays," preliminary product specification.

[GH] M. Gruber and K. Y. Hsu, "Moment-Based Image Normalization with High Noise-Tolerance," *IEEE Transactions Pattern Analysis and Machine Intelligence*, Vol. 19, No. 2, February 1997, pp. 136-139.

[GW] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 1992, pp. 1-18.

[Hauck] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *Proc. IEEE*, Vol. 86, No. 4, April 1998, pp. 613-638.

[HCS] Donald L. Hung, H. D. Cheng, and S. Sengkhamyong, " Design of a Configurable Accelerator for Moment Computation", *IEEE Transactions on VLSI*, Vol. 8, No. 6, December 2000, pp. 741-746.

[PT] J. Poldre and K. Tammemae, "Reconfigurable Multiplier for Virtex FPGA," *Proc. 9th Intl. Workshop Field-Programmable Logic and Applications*, 1999, pp. 359-364.

[PTVF] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 1992, pp. 610-615

[RAM] Xilinx, Inc. (2000) "Using the Virtex Block SelectRAM+ Features," Xilinx, Inc. application note XAPP130.

[RGS] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," *Proc. IEEE*, Vol. 81, No. 7, July 1993, pp. 1013-1029.

[SJV] B. Schoner, C. Jones, J. Villasenor, "Issues in Wireless Coding using Run-Time Reconfigurable FPGAs," *Proc. IEEE Workshop in FPGAs for Custom Computing Machines*, April 1995, pp. 85-89.

[TC] C. H. Teh and R. T. Chin, "On Image Analysis by the Methods of Moments," *IEEE Transactions Pattern Analysis and Machine Intelligence*, Vol. 10, No. 4, July 1988, pp. 496-512.

[VCC] Field Programmable Gate Arrays, Virtual Computer Corporation, *http://www.vcc.com*.

[VM] Xilinx, Inc. (1999), "Variable × Variable Multiplier RPMs for Virtex," application note.

[VMS] J. Villasenor and W. H. Mangione-Smith, "Configurable Computing," *Scientific American*, June 1997, pp. 66-71.

[WE] M. Wojko and H. ElGindy, "Configuration Sequencing with Self Configurable Binary Multiplier," *Proc. 6th Reconfigurable Architectures Workshop*, 1999, pp. 643-651.

[WH] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Using Run-Time Circuit Reconfiguration," *IEEE Transaction on VLSI*, Vol. 6, No. 2, June 1998, pp. 247-256.

[Wojko] M. Wojko, "Pipelined Multipliers and FPGA Architecture," *Proc. 9th Intl. Workshop Field-Programmable Logic and Applications*, 1999, pp. 347-352.

[WOLF] Digital Image Processing, Wolfram Research, *http://library.wolfram.com*.

Vita

Cheowway Neoh is from Sungai Petani, Kedah, Malaysia. He received his bachelor of science degree in May 2000 from Louisiana State University, majoring in electrical enginnering. Upon graduation, he continued with his master's degree which will be awarded at the Spring Commencement 2002.