Adaptive image filtering using run-time reconfiguration

Nitin Srivastava
Louisiana State University and Agricultural and Mechanical College

ADAPTIVE IMAGE FILTERING 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
Nitin Srivastava
B.Tech., Regional Engineering College,
Warangal, India, 1997
May, 2003

Acknowledgements

I wish to thank my major professors, Dr. Jerry Trahan and Dr. Suresh Rai, for providing me with constant guidance and inspiration throughout the entire period of my thesis work. During the initial phase, they helped me to understand the basic concepts and theories that were the building blocks for this research. Later on, their sound suggestions helped me shape my ideas into reality. This research work was a great learning experience for me, and I am grateful to them for providing me with an opportunity to work with them.

I wish to express my gratitude for Dr. R. Vaidyanathan who has been most helpful to me during the various phases of this thesis. His insight into my problems and his timely advice were very enlightening, which made it possible for me to overcome many problems.

Last but not the least, many thanks to Dr. David Koppelman, who was a great help, especially during the simulation and synthesis phase of this research work. His explanations about Leonardo and VSIM, and his advice about VHDL made it easy for me implement the design in an accurate and speedy manner.

This work was supported in part by the National Science Foundation under grant number CCR-0073429.

Table of Contents

Abstract

This thesis implements an adaptive linear smoothing image filtering algorithm, on a Virtex™-E FPGA using run-time reconfiguration (RTR). An adaptive filter uses a filtering window that runs over the entire image pixel-by-pixel, generating new (filtered) values of the pixels. As the name suggests, an adaptive filter can adapt to the varying nature of an image by adjusting the coefficients of the filtering window depending upon the local variance in the intensity values of pixels. It filters an image in a non-uniform fashion providing greater smoothing in largely uniform areas of the image and lesser smoothing when it encounters edges and step changes in the image.

These continual changes, in the coefficient values of the adaptive filter pose a problem in utilizing run-time reconfiguration (RTR) for its implementation, as benefits of RTR emerge only with considerable computing time between reconfigurations. This thesis provides a solution to this problem and reduces the running time of the algorithm through aggressive use of RTR.

This work provides details on the RTR implementation of an adaptive filter, along with an estimate of running time and hardware resource requirements, when synthesized on the Virtex™-E FPGA. We use a 3×3 size filtering window, and a 256×256 size gray scale image as a specific case, achieving speedup of 31 and 84 over pure software implementations running on Pentium III and Sun Ultra systems respectively.

Chapter 1: Introduction and Motivation

Digital image processing is an ever expanding and dynamic area with applications reaching out into our everyday life. Scientists from space exploration to forensic science have recognized digital or computer images as a powerful and efficient way of representing information. Computer images have gained prominence not only because they represent graphical data in an accurate form but also because computers can process them in a fast and efficient way.

A digital image comprises discrete elements called *pixels* arranged in rows and columns across the entire image. A pixel has an intensity value that is realized on screen when the image is displayed. For example, a pixel with low intensity value will appear darker on screen relative to a pixel with high intensity value. Such regular collections of pixels along with their intensity values form and define an image.

It is not very uncommon for intensity values of pixels of an image to change and acquire random values when an image is transmitted through communication channels or when a photograph generated by conventional cameras is digitized. This random intensity value of pixels is called *noise*. It is important to remove noise from an image to restore a digital image to its original form [IMG, TKP, LIM].

Needs of the modern world have dictated development and implementation of numerous algorithms to process computer images in various ways. Forensic scientists use applications that help them match fingerprints, while space scientists use applications that help them to solve the mysteries of outer space. All these applications work using the same basic methods of digital image processing to process digital images. One such method that is used to *denoise*, that is, remove noise from a digital image and thus restore the image to its original form is *image filtering* [JKS].

A host of algorithms have been developed to achieve this objective. One such algorithm employs a *linear smoothing filter* that uses a square mask containing coefficients arranged in rows and columns. The filter runs the mask over the entire image to correct anomalies in intensity values of pixels [JKS]. Chapter 2 provides a detailed working of a linear smoothing filter.

A common linear smoothing filter called a *spatially invariant linear smoothing filter* uses coefficients whose values remain the same for every position of the mask over the entire image. If the image being filtered is non-uniform in nature, then a spatially invariant filter can blur the image. This is because a spatially invariant filter does not adjust the values of its coefficients according to the nature of the image. For example, the linear invariant filter will filter the pixels representing edges in an image in the same way as pixels representing uniform areas in the image. This can lead to edges appearing fuzzy in the filtered image. Furthermore, some areas of the image may require less smoothing than others, depending on the noise ratio of the respective areas. This non-adaptive nature of a linear spatially invariant filter makes it unsuitable for filtering non-uniform images.

A variety of linear smoothing filter called a s*patially varying linear smoothing filter* or *adaptive linear smoothing filter* performs better than a spatially invariant smoothing filter as the values of its coefficients can change across the image and it can adapt to the varying nature of the image. This helps to remove noise and maintain details within an image that are not possible with a spatially invariant filter. This thesis implements an adaptive image filtering algorithm [JKS, LIM].

Field-programmable gate arrays (FPGAs) are programmable (reconfigurable) devices that permit us to implement different hardware designs by *reconfiguring* (programming) them over and over again. This feature is not available on non-reconfigurable hardware. For example, a general-purpose microprocessor has a fixed number of instructions that execute on static hardware. It is not possible for this set of instructions or the underlying hardware to change as per the specific requirements of an application. This can lead to poor efficiency resulting in greater running times for some applications. FPGAs, being reconfigurable, overcome this limitation, as we can reconfigure them to provide application specific hardware, which of course is more efficient than the static general-purpose hardware of microprocessors.

In many applications, the hardware requirement is more than what is available on FPGAs. *Run-time reconfiguration* (RTR) is the concept of breaking the entire flow of an algorithm into phases, providing a specialized hardware design for each phase by reconfiguring the FPGA or a part of it (on partially reconfigurable FPGAs described in Section 2.1.2.3). We can swap different phases in and out of the FPGA as per the execution of an algorithm. This is cheaper than a general-purpose design for the algorithm as different phases can use the same hardware resources. It also makes the design faster as each phase executes on specialized hardware. Furthermore, it is possible for more than one phase to execute concurrently. This reduces the running time of an algorithm considerably. RTR has reduced the running time of many algorithms considerably [CH, HCK, VH, HW].

We implement an adaptive linear smoothing filter using RTR in this thesis. Our motivation to use RTR stems from the fact that many real-time applications that process digital images need faster implementations of image filtering algorithms to meet their strict timing constraints. We use a Xilinx Virtex™-E FPGA as it is fast and partially reconfigurable, that is, new data can be loaded and configured on the device without stopping the application [XIL01, XIL04].

Three chapters follow this chapter. Chapter 2 provides information on FPGAs, RTR, and image filtering concepts. It ends with a discussion of prior related work. Chapter 3 provides details on the implementation of a 3×3 adaptive filter on a Xilinx Virtex™-E FPGA for a 256×256 size gray scale image. Chapter 4 reports simulation and synthesis results, that is, the running time of the algorithm and hardware requirements for the design.

Chapter 2: Background

This chapter provides information on the basic concepts of field-programmable gate arrays (FPGAs) including a description of the Xilinx Virtex™-E FPGA (the FPGA used to implement our design), run-time reconfiguration (RTR), and the adaptive image filtering algorithm implemented in this thesis. These concepts are fundamental to the understanding of the present work detailed in the following chapters. The chapter concludes with a discussion of prior related work.

2.1 Field-Programmable Gate Arrays (FPGAs)

An FPGA is a programmable device constructed basically of three kinds of elements: *configurable-logic blocks* (CLBs), *input/output blocks* (IOBs), and *interconnection*. A CLB can be programmed to realize different combinational and sequential logic functions. The interconnection consists of wire segments of varying lengths that can be connected together by means of programmable switches. They serve to connect a number of CLBs together to realize a design. The ability to program a CLB over and over again and the flexibility of interconnection between CLBs make an FPGA an ideal device for implementing and testing ASIC prototypes. Figure 2.1 shows a generic FPGA architecture. Figure 2.2 shows a detailed view of CLBs with routing resources for interconnection. FPGAs generally have complex routing architectures and dense interconnection making it is possible to implement complex designs on FPGAs in contrast to traditional programmable logic devices (PLDs). Traditional PLDs use two-level AND-OR logic gates with wide input AND gates to implement logic while FPGAs typically use multiple levels of lower fan-in gates. This makes an FPGA compact and more efficient than PLDs.

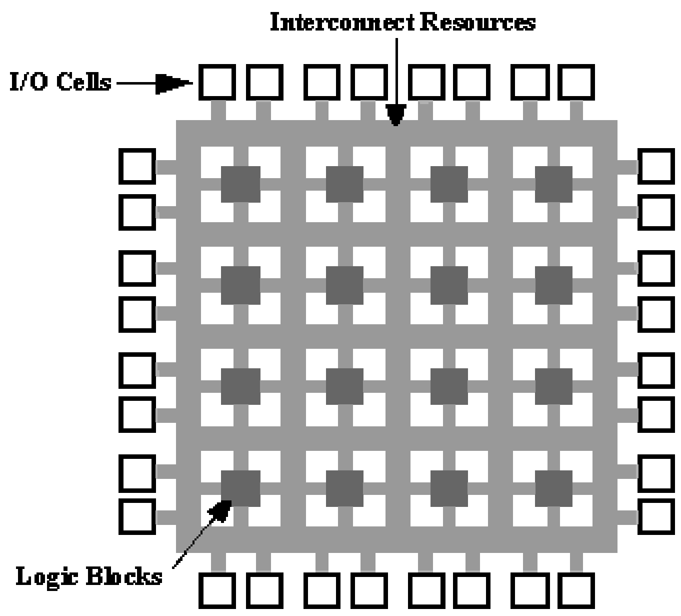

Figure 2.1 - Generic FPGA architecture [VCC]

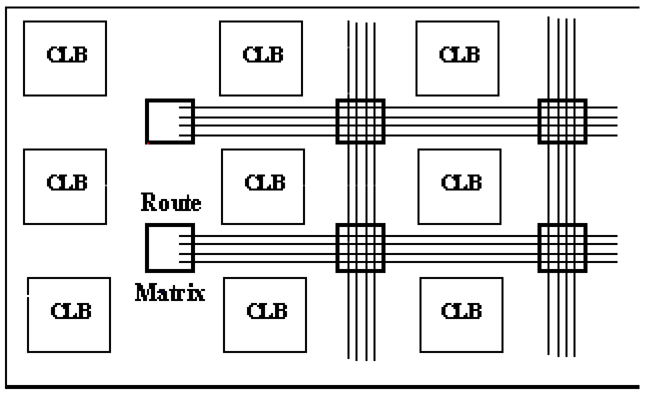

Figure 2.2 - CLBs with interconnection [VCC]

An FPGA can theoretically contain CLBs as complex as a microprocessor or can be as simple as a transistor, though commercial FPGAs typically have CLBs based on transistor pairs, two input NAND gates, multiplexers, or look-up tables (LUTs).

Commercial FPGAs are categorized into four major classes based on their interconnection and the way they can be programmed. The interconnection can be symmetrical array, row-based, hierarchical, or sea-of-gates (Figure 2.3).

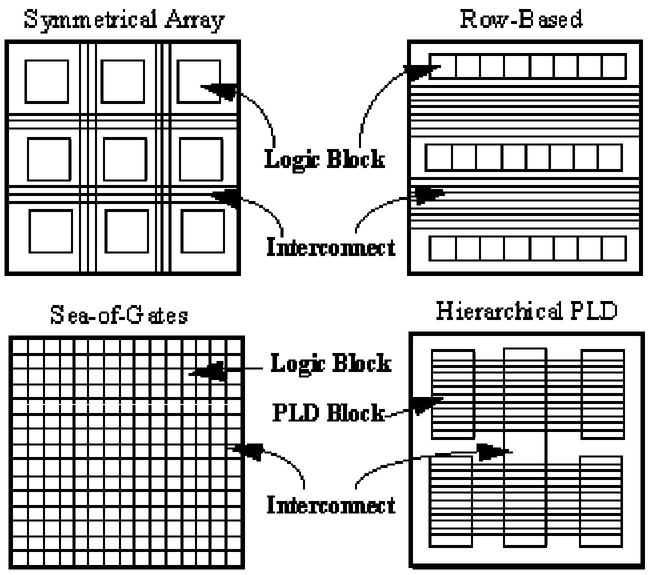

Figure 2.3 - Different interconnections for FPGAs [VCC]

Commercial FPGAs can use four types of programming technology to program the FPGA. They are Static RAM (SRAM), anti-fuse, EPROM, and EEPROM technologies. These technologies have their merits and demerits, and the choice of an FPGA depends upon the type of design to be implemented, For example, SRAM technology makes it possible to reprogram the connections but needs larger space. Anti-fuse technology is less expensive, but can be programmed only once. EPROM/EEPROM technology provides features to reprogram the FPGA, but FPGAs using EPROM cannot be reprogrammed in-circuit. It is possible, however, to reprogram SRAM and EEPROM-based FPGAs in-circuit [HCK, VH, HW, VCC, RGV, BR]. Table 2.1 compares features of four commercially available FPGAs.

Table 2.1 - Comparison of four 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

2.1.1 Fine and Coarse Grained FPGAs

Based on CLB size, we can classify FPGAs broadly into two types, *fine grained* and *coarse grained.* Fine grained CLBs are smaller in size and do not possess the capability to individually implement complex logic functions. Though their smaller size makes it easier to use them efficiently by better utilizing the hardware resources, they also need a large number of wire segments and programmable switches to connect them. Thus, FPGAs containing fine grained CLBs are less dense and also slower, as the wire segments take longer time to pass data from one CLB to another due to the greater number of programmable switches required. *Crosspoint* and *Plessey* FPGAs contain fine grained CLBs.

A *coarse* grained CLB is more complex in nature. FPGAs produced by Xilinx, Altera, and Actel employ *coarse* grained CLBs. Coarse grained CLBs need fewer wire segments and fewer programmable switches to connect them together, thus resulting in denser and faster

FPGAs. As the CLBs become larger, however, it becomes difficult to use the hardware resources on the FPGA efficiently. Choice of a particular FPGA thus depends largely on the space and timing requirements of the application to be implemented [RGV, HCK].

2.1.2 Virtex™-E FPGA

We use the Virtex™-E FPGA produced by Xilinx, Inc. for our implementation. The Virtex™-E FPGA architecture has two major components, CLBs and IOBs. The Virtex™-E FPGA also has dedicated block memories called *Block SelectRAM*™ memories (BRAMs). The Virtex™-E belongs to the Virtex™ family of FPGAs which features regular arrays of CLBs arranged in columns surrounded on all sides by IOBs (Figure 2.4). The interconnection within them is very versatile as the wire segments are of varying lengths and the programmable switches are fast and placed in locations that allow them to efficiently connect these wire segments. Virtex™ FPGAs are SRAM-based. We can implement a design by loading configuration data into their internal memory cells. The values stored in dedicated static memory cells define the configuration of CLBs and their interconnection.

Interconnection of CLBs is through a *general routing matrix* (GRM) shown in Figure 2.5. The GRM contains routing switches that connect the vertical and horizontal routing channels. Each CLB nests into a *VersaBlock*™ that connects the CLBs to the GRM [XIL01, XIL02].

2.1.2.1 Configurable Logic Block (CLB)

A Virtex™-E CLB contains four *logic cells* (LC). An LC is the basic building block of the CLB. An LC contains a 4-input function generator, carry logic, and a storage element. The entire CLB is made of two CLB slices, each containing two LCs. Figure 2.6 illustrates the various components of the Virtex™-E CLB.

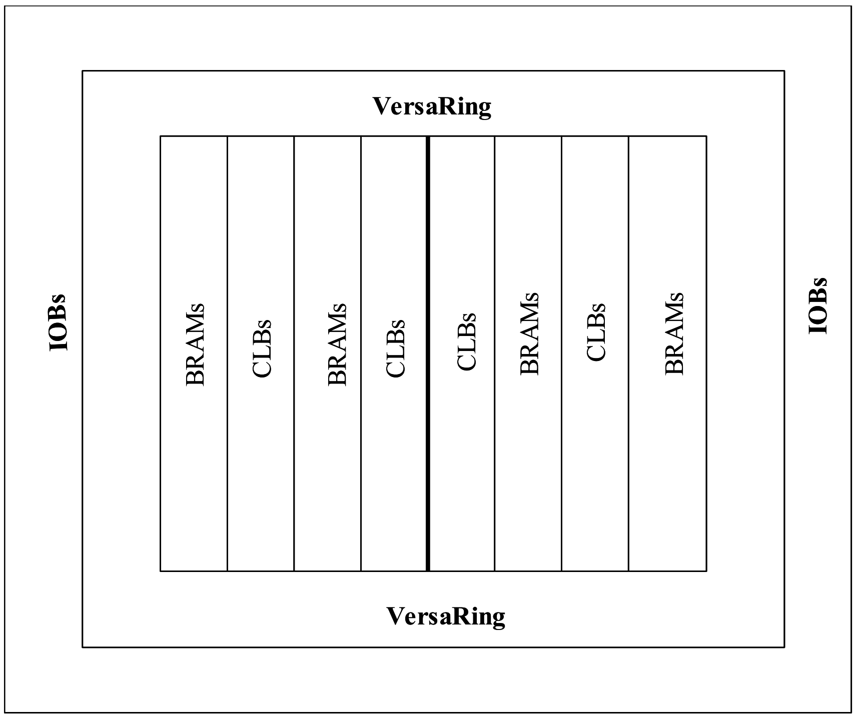

Figure 2.4 - Virtex™-E architecture overview [XIL01]

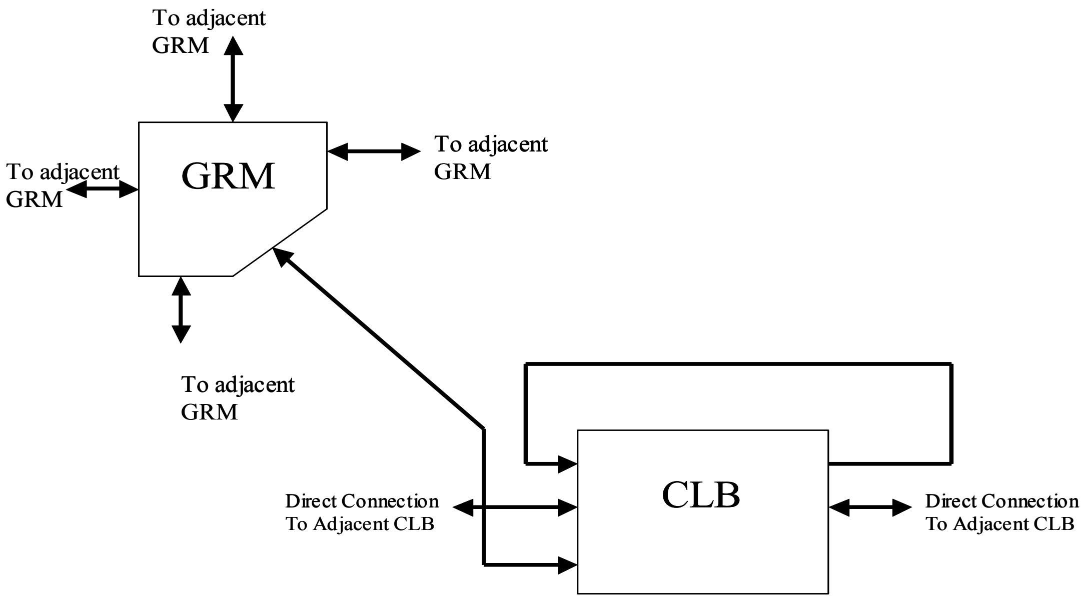
To adjacent GRM
GRM
To adjacent GRM
To adjacent GRM
To adjacent GRM
CLB
Direct Connection To Adjacent CLB
Direct Connection To Adjacent CLB

Figure 2.5 - Virtex™-E routing architecture [XIL01]

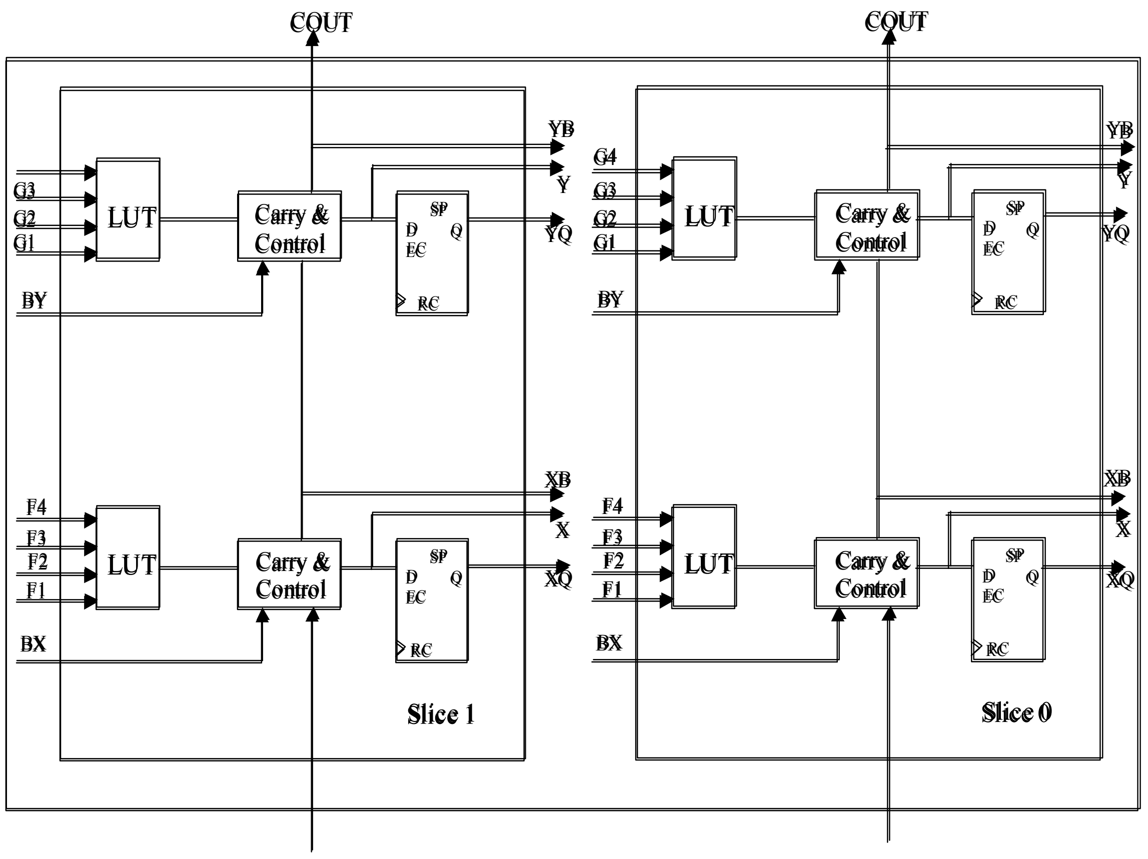

Figure 2.6 - 2-slice Virtex™-E CLB [XIL01]

Four-input *look-up tables* (LUTs) with 16 locations in each LUT implement function generators. We can implement a function in an LC by loading data into the LUT. The input into the LC is an address into the LUT. The value stored at that address is the output of the LC. We can combine the two LUTs per slice to provide functions of five or six inputs. Each LUT can also work as a 16×1-bit synchronous RAM.

2.1.2.2 Block SelectRAM™ Memory (BRAMs)

Block SelectRAM™ memories (BRAMs) are dedicated blocks of memory that can store large amounts of data. Each memory block is four CLBs high and is organized into memory columns stretching the entire height of the chip. There is one such memory column between every twelve CLB columns. Each Block SelectRAM™ is dual ported and can store 4096 bits. The width of each addressable location can vary from 1 to 16 bits. For example, if each location is 16-bits wide, then we have 256 such locations within one Block SelectRAM™ memory [XIL01, XIL03]. We have *block memories*, using Block SelectRAMs™ in our implementation to store partial results of our computation.

2.1.2.3 Partial Reconfiguration

The Virtex™-E class of FPGAs provides the facility to load new configuration data into a portion of the FPGA while the rest of the FPGA is actively computing. Our choice of using a Virtex™-E device is to some extent guided by this feature. In using run-time reconfiguration (explained in the next section) we need to reconfigure some portions of the FPGA with new data while other portions continue computing. This reduces the running time of the algorithm and achieves much higher speedups than possible without the ability of the FPGA to partially reconfigure itself [XIL04].

2.2 Run-Time Reconfiguration (RTR)

We can allocate hardware resources on an FPGA statically or dynamically. In static allocation, the entire application resides on the FPGA for the entire running time of the algorithm. No hardware allocation or reconfiguration takes place while the application is running. This is called *compile-time reconfiguration* (CTR). Because of its similarity with traditional designs, most current FPGA applications use CTR [HW].

Run-time reconfiguration (RTR), as the name suggests, is a concept that allows parts of the design to be configured with new data during the course of a computation. RTR aims at reducing both hardware requirements as well as computation time for an application as it uses the same hardware resources multiple times, and applies specialized hardware for each phase of an application. Each application that uses RTR consists of multiple configurations with each configuration implementing some fraction of the application. An individual configuration is a *configuration context*. The process of switching between configuration contexts is called a *configuration context switch* [WE]. In Chapter 3, we define a configuration context in a specific way related to our design.

2.2.1 Global and Local RTR

We can implement RTR as *global* RTR or *local* RTR. Global RTR means allocating all the available hardware resources to each configuration context. The application stops to load each new configuration context and then restarts. It is difficult to break an application into portions that have equal hardware resource requirements; so global RTR may lead to wastage of resources. As an advantage, global RTR can use conventional CAD tools successfully for each separate configuration [HW].

Local RTR on the other hand means loading a new configuration context onto a part of the FPGA without stopping the remainder of the application. Local RTR uses hardware resources more effectively as it does not configure the entire hardware resource for each phase of the application. We utilize local RTR (henceforth called RTR) in our implementation and use the partial reconfiguration feature of Virtex™-E FPGA to implement it. Since the application does not need to be stopped to load each new configuration context, computation and reconfiguration times can overlap, drastically reducing the running time of our application.

2.2.2 Constant Coefficient Multiplier (KCM)

It is important here to describe the way we implement RTR in our design. The reconfigurable part of our design is a set of *constant coefficient multipliers* (KCMs). The remainder of the design is fixed, that is, it does not change on every configuration context switch. As shown in Figure 2.7, we configure one set of KCMs (context) while the other set is operating. We provide details about our circuit and reconfiguration method in Chapter 3.

A KCM comprises look-up tables (LUTs) and adders. We use 8-bit KCMs in our design as shown in Figure 2.8 for constant *k,* to produce the 16-bit product of an 8-bit input and an 8-bit constant. The LUTs store 16 results ranging from 0 through 15 times the constant value *k*. We break the 8-bit multiplier input into two 4-bit values, each addressing a different LUT to produce two 12-bit values (the product of the 4-bit input and the 8-bit constant *k*). The 12-bit outputs combine to produce the final 16-bit result. Configuring a KCM means loading new values into its LUTs to correspond to a new multiplier constant [XIL04, WE]. Please note that it takes 16 clock cycles to reconfigure a KCM because an LUT has 16 locations within it and it takes one cycle to load data into each location.

2.3 Image Filtering Concepts

This section describes the image filtering algorithm that this thesis implements. Presence of *noise* corrupts an image. Presence of *salt & pepper noise* in an image results in occurrences of both black and white intensity values, while *impulse noise* introduces pixels of white intensity only. *Gaussian noise* results in changes in the intensity values of the pixels. An image filtering algorithm performs the task of removing noise from an image. The image in consideration here is a gray scale image with pixel intensity values ranging from 0 (darkest) to 255 (brightest). A filtering algorithm works on the principle that any pixel having an intensity value very much different from its surrounding pixels is noisy. It is the objective of a filtering algorithm to compute new values for each pixel taking into account the intensity values of its surrounding pixels. A good filter used to remove noise from an image is a *linear smoothing filter*. Figure 2.9 shows on the left a gray scale image containing 20% salt & pepper noise and on the right the image smoothed by a linear smoothing filter.

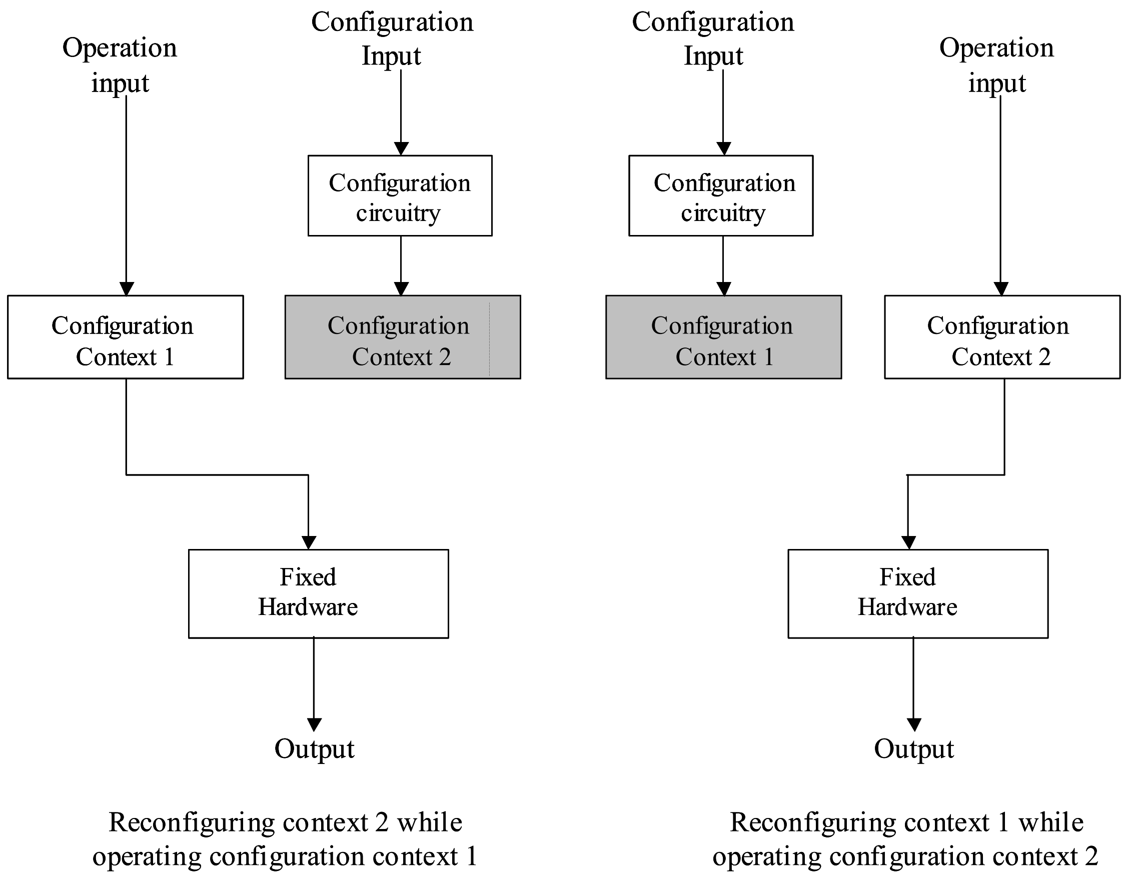

Figure 2.7 - Reconfiguring one context while the other is operating [WE]

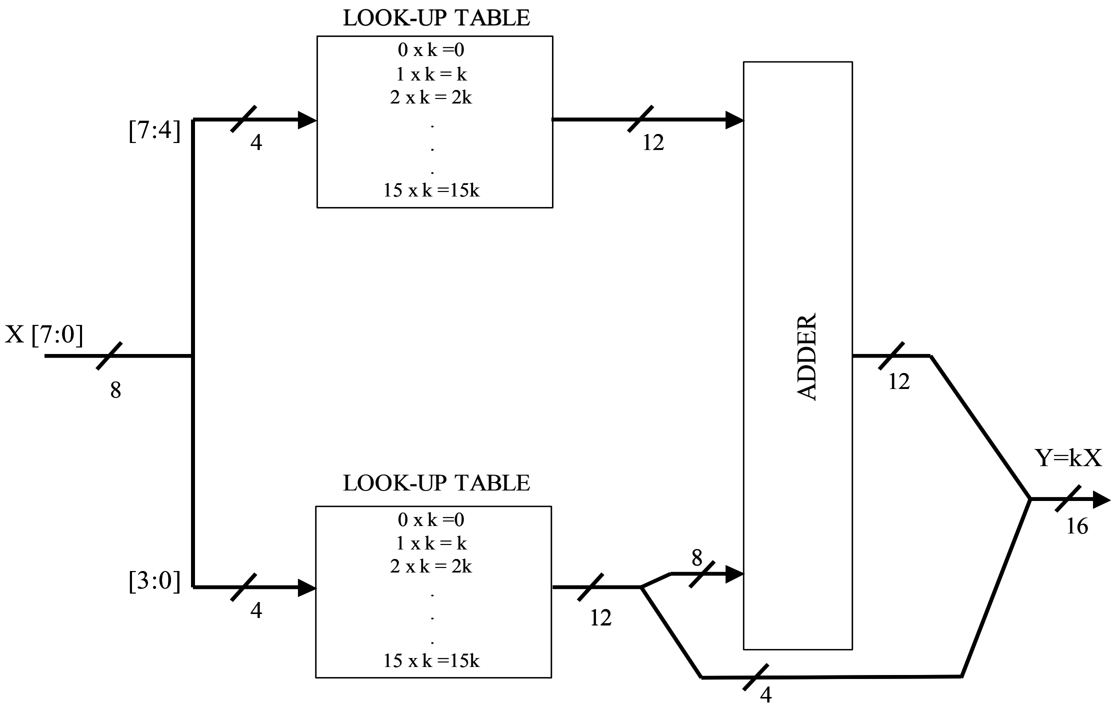

Figure 2.8 - 8-bit constant coefficient multiplier (KCM) [XIL05]

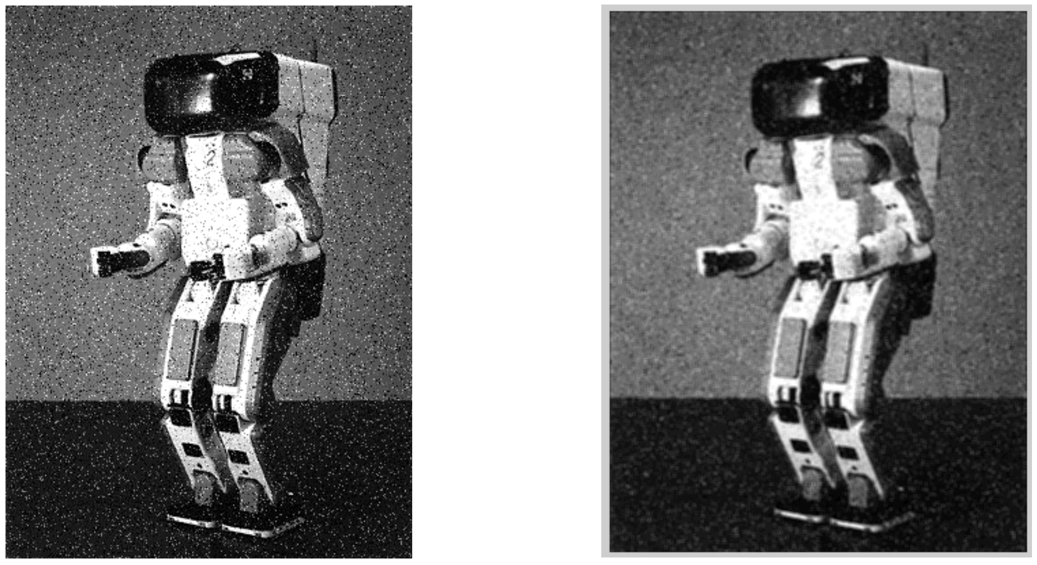

Figure 2.9 - An example noisy gray scale (left) image smoothed by a linear smoothing filter (right) [IMG]

The image filter under consideration is a smoothing filter because it smoothes out the noise present in the image by distributing the intensity of a noisy pixel among its neighboring pixels by averaging the pixel intensity values. It is actually a *filtering window* that moves over an image pixel by pixel. The filter multiplies the intensity values of pixels it overlaps with its coefficients and sums the products together to produce the new value of the pixel at which it is centered. Figure 2.10 shows the working of a linear smoothing filter using a 3x3 size filtering window. It is linear in nature as the new value of a pixel is the weighted sum of the intensity value of all pixels overlapped by the filtering window.

The filtering window moves over an image pixel-by-pixel starting from the top left corner to the bottom right corner of the image. It shifts over one pixel column at a time until the end of a row of the image and then shifts down by one pixel row. At any given position within the image, the filtering window overlaps a certain number of pixels depending upon its size. Filtering windows are typically of size 3×3, 5×5, or 7×7.

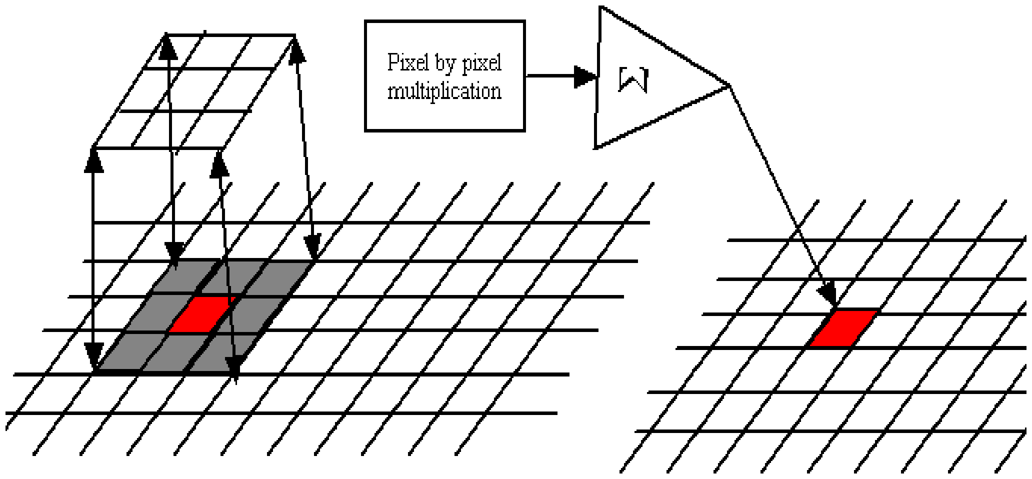

Figure 2.10 - Working of a linear smoothing filter [IMG]

We can represent the working of a linear smoothing filter of size $w \times w$ by the following expression:

$$nv[i,j] = \sum_{g=-(w-1)/2}^{(w-1)/2} \sum_{h=-(w-1)/2}^{(w-1)/2} v[i+g, j+h]*cv[i,j,g,h], \tag{2.1}$$

where $v[i,j]$ denotes the intensity value and $nv[i,j]$ denotes the new value of a pixel at position $[i,j]$ in the image, where i is the row number and j is the column number; and $cv[i,j,g,h]$ is the value of the filtering window coefficient at position $[g,h]$ within the filtering window for pixel $p[i,j]$ where the center of the filtering window is [0,0].

Linear smoothing filters can be of two types. If the filter coefficients remain the same at all positions of the filtering window over the image, then it is a *spatially invariant smoothing filter.* This filter removes noise from the image, but it can also blur the image as sharp edges are smoothed and step variations occur as gradual changes. The second type is a *spatially variant linear smoothing filter* or *adaptive filter* in which filter coefficients adapt to the varying nature of the image and can be different for different positions of the filtering window over the image. Such a filter can adjust the values of its coefficients to perform less smoothing near the edges and to perform more smoothing in areas where the image is largely uniform in nature and thus preserves the details in the image [JKS, IMG, TKP, LIM]. This thesis implements an adaptive filter. Section 2.4 discusses one method to generate coefficients for an adaptive filter. Our implementation receives filtering window coefficients as input rather than generating them itself, so it can work with any scheme for generating the window coefficients.

The smoothing filter does not smooth the pixels occurring at the image boundaries. The number of rows and columns not filtered at each image boundary is equal to $(w\text{-}1)/2$, where $w \times w$ is the filter size. For example, if the filter is of size 3×3, then the pixels in the top and bottom rows and left and right columns of the image are not filtered.

2.4 Generating Coefficients for an Adaptive Filter

Tekalp [TKP] discusses one approach to generating coefficients for an adaptive filter. This thesis does not implement any means of generating coefficients on the FPGA, though a solution to this problem can be a worthwhile addition to our implementation. Though this approach emphasizes generating filter coefficients to denoise video images, we can adapt it for the case of two-dimensional gray scale images. We compute the coefficient values based on the uniformity of the image where the coefficients are of equal weights when the image is uniform. When the intensity values of pixels overlapped by the filtering window are very different from the intensity value of the pixel to be filtered, the coefficient acquire values to provide greater weightage for pixels whose intensity values are nearer to the intensity value of the pixel to be filtered. This requires optimizing a criterion function, which depends upon the intensity values of the pixels overlapped by the smoothing filter.

We first calculate a normalization constant, *K*, for each pixel, which provides information about the variation in the intensity values of pixels in its $w\times w$ neighborhood, where $w\times w$ is the size of the filter and $-(w-1)/2\leq g,h \leq(w-1)/2$. The normalization constant *K* for each position of the pixel can be calculated as follows.

$K[i,j]$ =

$$\left(\sum_{g=-(w-1/2)}^{(w-1)/2}\sum_{h=-(w-1)/2}^{(w-1)/2}\frac{1}{1+a*\max\left\{\varepsilon^2,\left[v[i,j]-v[i+g,j+h]\right]^2\right\}}\right)^{-1} \qquad (2.2)$$

where ε and *a* are constants. We use the normalization constant *K* to calculate the value of a coefficient *cv*[*i,j,g,h*] for the filter centered at position [*i,j*] within the image as follows.

$$cv[i,j,g,h] = \frac{K[i,j]}{1 + a * \max\left\{\varepsilon^2, [v[i,j] - v[i+g, j+h]]^2\right\}} \tag{2.3}$$

If the square of the difference between the intensity value of a pixel and its neighboring pixels is smaller than the constant ε, that is, the image is uniform in the neighborhood of the pixel being filtered, then all coefficients have the same value and the filter provides uniform smoothing. When the square of the difference between the intensity value of a pixel and its neighboring pixels is more than the constant ε, that is, the image is not uniform in the neighborhood of the pixel being filtered, then the coefficient weights within the filter are different.

Lim [LIM] has discussed two other approaches to filter an image in an adaptive manner. The first approach is to divide the image into sub-images and process each sub-image by a spatially invariant smoothing filter where the filter coefficients do not vary within the sub-image but can vary from one sub-image to another, thus adapting to the global intensity variations within the image.

The second approach involves changing the size of the filtering window to accommodate variance in the intensity values of pixels. In this approach, using a smaller size window in regions with large local variance helps to preserve the details of the image. The author uses larger size windows in areas where the image is more uniform in nature to provide better smoothing.

Other approaches to adaptive image filtering such as using a Noise Adaptive Soft-Switching Median Filter [EM] mostly employ non-linear filters to smooth the image.

2.5 Prior Related Work

This section discusses prior research in the area of run-time reconfiguration as well as work done in the area of image filtering.

Wojko and ElGindy [WE] looked into the use of RTR for the IDEA encryption algorithm and adaptive FIR filtering. Both applications have a common thread between them that makes them suitable for the use of KCMs. In both applications, one input to a multiplier changes frequently while the other remains constant for a set number of cycles. This inherent feature of the algorithms creates a natural home for KCMs. The authors used the slow changing input, as a fixed multiplier constant configured in the KCM that then was changed when required using reconfiguration. This approach makes the logic implementation smaller and faster than using general-purpose multipliers. They have used RTR aggressively to reconfigure new constant multiplier values into the KCMs.

IDEA uses six 16-bit sub-key sequences selected from a 128-bit encryption key. These values remain constant during one round of computation and hence the authors used them as the multiplier constant within the KCMs, providing the data to be encrypted as input to the KCMs. During this time a second set of KCMs is configured with new 16-bit sub-key sequences to be utilized during the next round of computation. They maintained the timing of reconfiguration such that as soon as all the data to be processed in the present computation round passes through a particular KCM, the KCM enters its reconfiguration phase. Thus, each KCM starts and finishes its reconfiguration phase at different times. This is an example of rolling reconfiguration where not all reconfigurable elements of the design are reconfigured simultaneously but one after another.

An FIR filter computes the dot product between a series of time samples and a weighted coefficient vector. The filter consists of taps, each tap multiplying one coefficient of the vector by all the input samples. The authors observed that each input sample resides within the filter for a number of cycles equal to the length of the coefficient vector, which is a fixed constant number of cycles, while the vector coefficients can change at arbitrary times. The authors therefore configured the KCMs with input samples and passed around the filter coefficients in a circular fashion so that each filter coefficient is multiplied in turn by each input sample. The design has two KCMs per tap, whereby one KCM can be reconfigured in time equal to or better than that for which the other KCM is active. This reduces the running time of the algorithm. As an input sample arrives, the system configures a KCM for one tap with the sample as constant. The system uses the next input sample to configure the KCM for the next tap in the filter and so on. The two KCMs per tap alternate between reconfiguration and active phases. By the time one KCM per tap processes all the coefficients, the other KCM completes reconfiguration. At this point, they exchange their roles, and the active one enters reconfiguration phase and the newly reconfigured KCM enters its active phase. The filter coefficients can also be updated over time by using an interface provided for this purpose. Various observations with different sized FIR filters proved that the application hardware requirements without reconfiguration are about 25% to 45% higher than with reconfiguration. The approach used to implement an adaptive filter in this thesis bears some similarities with this approach. As in the case of an adaptive filter, it is possible for filter coefficients to change rapidly and randomly; instead of input samples, we configure the KCMs with pixel values as constants as they remain constant for nine clock cycles (for a 3×3 size filter). In the case of a spatially invariant filter, though, the reverse approach is better, that is, using filter coefficients as constants within the KCMs, as filter coefficients remain constant throughout the run of the algorithm.

Key-specific DES is another application that benefits through the use of RTR. As each end user of a DES session shares the same secret key, Leonard and Mangione-Smith [LS] generate key-specific circuitry. This improves the speed of the circuit as a generic DES circuitry is complex and the routing complexity of a design reduces the speed of a design. Generating a design only for a specific DES key used for a particular session reduces the routing complexity of the design, resulting in a faster circuit. This approach is called *partial evaluation.* Since the session key remains static for long periods of time, the authors generated the sixteen sub-keys once and use them for long periods of time by using a multiplexer to select one of them. Thus, they used prior knowledge of the session key to tailor the encryption circuit, and thus reduced the hardware requirements by as much as 45% as opposed to a generic DES circuit. They employed RTR to reconfigure new values into the design of the encryption engine as the session key changes from one session to another.

Another example that signifies the power of RTR is its use in motion estimator applications [TBW]. Estimating the motion of an object in space involves processing the image by different algorithms, namely gaussian and averaging filters followed by temporal and spatial derivatives. Receiving images at a rate of 25 per second imposes the requirement that all algorithms run in real time to correctly estimate the motion trajectory of an object in three-dimensions. The authors used RTR to configure one portion of the FPGA with the implementation circuit of an algorithm while some other algorithm is processing data. This approach allows the images to be processed within the strict time limit of 40 ms.

Shirazi *et al*. [SLBC] used RTR to design a database search engine. Database search engines use a hash function to map a word to a pseudo-random value, which addresses into a look-up table (LUT), which indicates whether the word exists in the user dictionary or not. To create the user dictionary, the authors first hashed the words and configured the values generated into the LUT. This example is very suited to FPGA implementation as many commercial FPGAs, such as the Virtex™ family of FPGAs from Xilinx, use LUTs as basic elements in their CLBs. Shirazi *e. al.* used RTR to change parameters for the hashing functions, such as mask and shift values, at run time. RTR proved effective when switching between different hashing functions. Tests performed assumed three cases of different amounts of temporary memory available to the application and three different circuits to implement the circular shifter used to generate hash values of the input words. The results reported the time/area trade-offs in different approaches and suggested using these approaches for different timing and hardware requirements.

Adapting reconfigurable hardware to general purpose computing requirements has been a serious research area as there is lack of automatic mapping techniques to map traditional processor pipelines onto FPGAs. Bondalapati and Prasanna [BP] investigated the issue of mapping loop computations from applications onto high performance pipelined configurations. The statements are first executed on one stage of the pipeline during which the next stage of the pipeline is configured at run time to execute the statements through the next stage of execution. Experiments with *N*-body simulation and an FFT algorithm reported speed-ups of 2.74 and 6.38, respectively, as opposed to their running times on traditional microprocessors. Some other applications like parallel object recognition [CCP] and acceleration of pipelined integer and floating-point accumulations [LM], though they do not use RTR, gain considerable speedups when implemented on FPGAs as compared to software-based approaches.

Chapter 3: Implementation

This chapter provides a detailed account of the implementation of the adaptive filtering algorithm (as discussed in Chapter 2) on a Xilinx Virtex™-E FPGA. We discuss implementation details for a 3×3 size filtering window on a 256×256 size image and the way we utilize the concept of run-time reconfiguration offered by FPGAs. The chapter starts with a description of the computation subsystem and then moves on to discuss the working of a 1×3 size filtering window followed by a description of the working of the full 3×3 size filtering window. We discuss other subsystems (I/O and memory) later in the chapter. Lastly, we discuss the boundary handling subsystem that handles pixels at image boundaries.

3.1 Computation Subsystem

The circuit for this implementation is hierarchical and is described in the same fashion. The basic component is a *module* (Figure 3.1). Sixteen modules connect together with a pipeline register between each pair of adjacent modules, as shown in Figure 3.2, to form the *computation subsystem.* Because the image contains 256 pixels per row, by choosing a multiple of two as the number of modules in the computation subsystem, all pixels processed at the same time belong to the same row. We number the modules 0 through 15. We assume that the pixel values are received in row major order, that is, from the top left corner of the image to the bottom right corner. For a particular module, its *previous module* is the module from which it receives data and its *next module* is the module to which it sends data. For example, for module 3, module 2 is its previous module and module 4 its next module. For module 0, its previous module is module 15, and for module 15, its next module is module 0.

Additional elements in the design provide the required routing paths between the I/O pins of the FPGA and the modules. We also have image boundary handling circuits for pixels occurring on the boundaries of the image, as these pixels are not filtered. In this section we describe only the different elements and their interconnections. Later sections describe how the data flows through them and the control of data through various stages.

A module comprises a number of separate entities (Figure 3.1). These are two 8-bit KCMs, a 2×1 multiplexer called *KCM output mux*, a 19-bit adder called *module adder*, a 4-bit modulo-up counter called *step counter*, a register that holds a constant value of zero called *zero register*, a 3×1 multiplexer called *module mux*, two 16-bit registers called *memory write register* and *module read register* connected to the write and the read ports of the block memory (refer to Section 3.6), respectively.

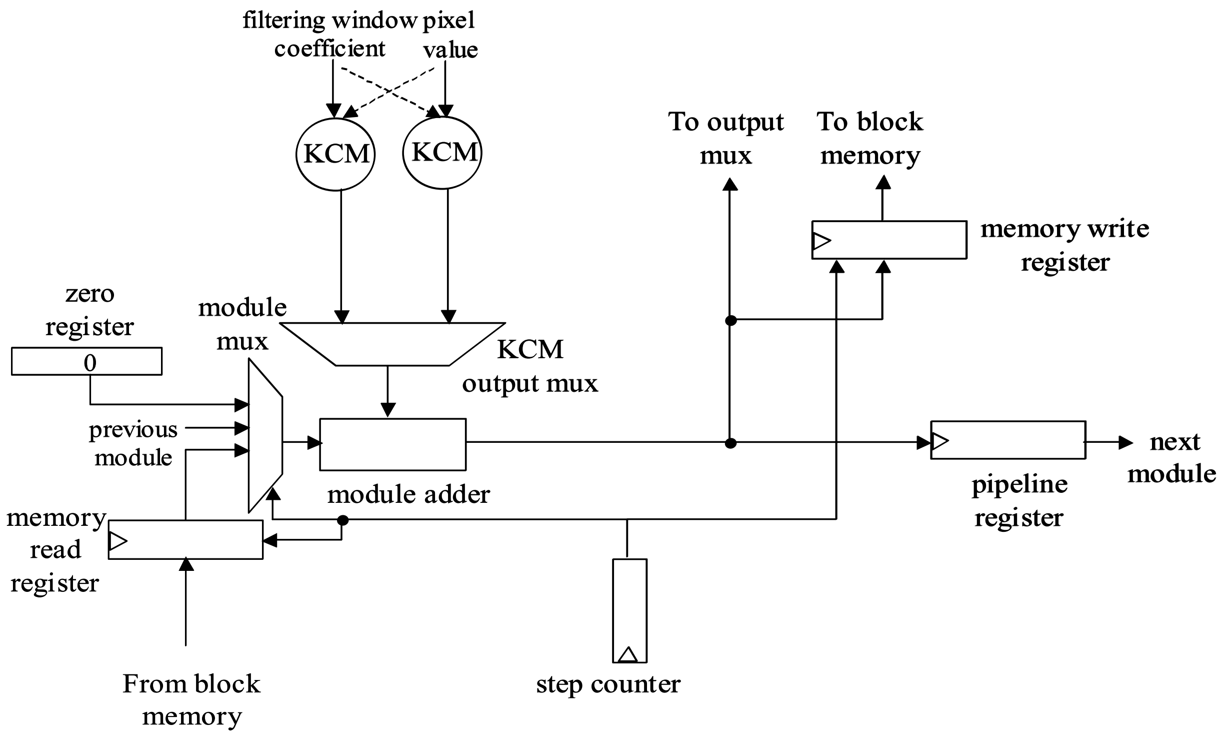

Figure 3.1 - Circuit layout of a module

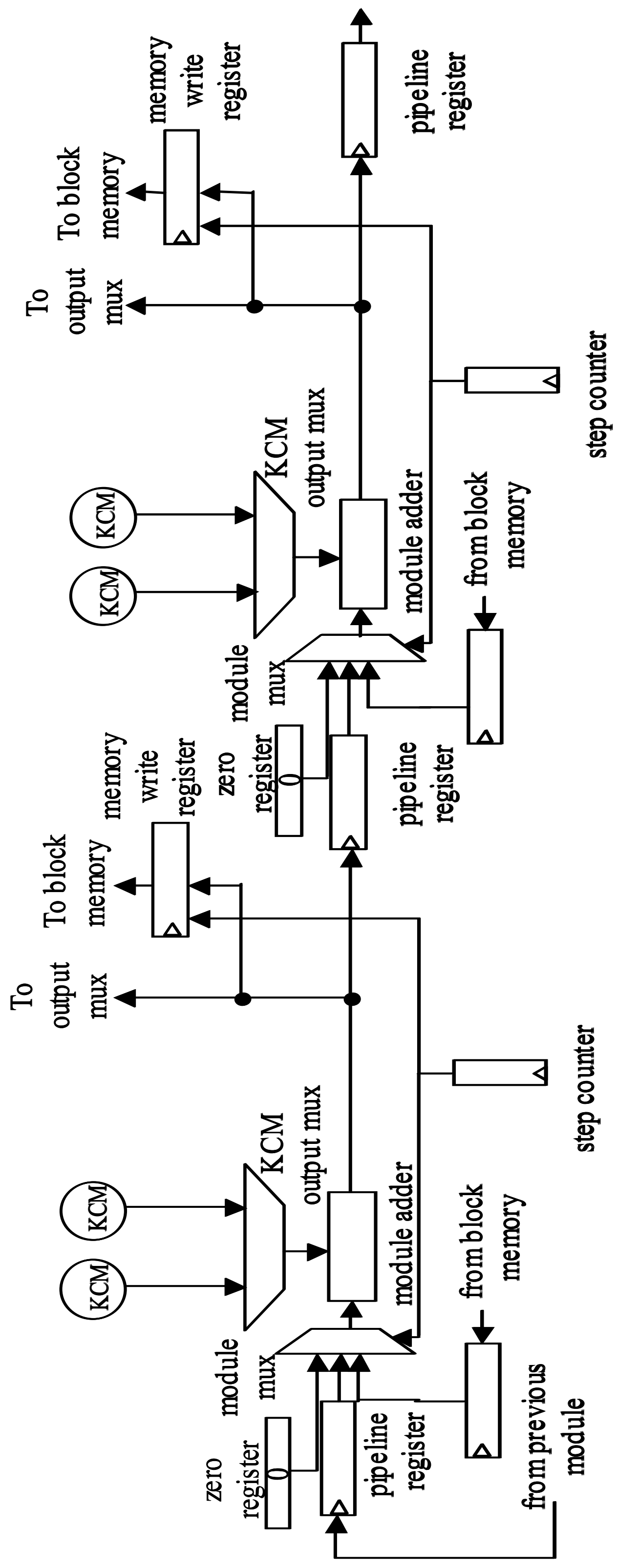

Figure 3.2 - Two modules connected together in the computation subsystem

The presence of two KCMs is the key to run-time reconfiguration as one KCM can provide data to the module adder while the system is reconfiguring the other one. Each KCM receives the filtering window coefficient as input (recall that the KCM is already configured with a value of an image pixel) and produces a 16-bit value (product of filtering window coefficient and pixel value) that it feeds to the module adder. The KCM output mux selects the output of the active KCM to pass to the module adder. The configuration context counter (described in Section 3.5) provides the select signal to the KCM output mux. The other input to the module adder comes from the module mux. The module mux has three inputs, the first connected to the zero register, the second to the pipeline register connecting the module to its previous module, and the third to the memory read register. The step counter counts up by one on every rising edge of the clock and rewinds to zero after reaching a count of 15. All modules in the computation subsystem work in parallel, and data moves along the same path within each module, so outputs appear simultaneously on the same output port of each respective module.

It is important to introduce at this stage the concept of a *configuration context.* Every KCM alternates between computation and reconfiguration phases. At any time the set of 16 KCMs in their computation phase (one per module) is called the *active set*, while the other set of 16 KCMs in their reconfiguration phase (one per module) is called the *reconfiguring set*. The computation subsystem with the active set of KCMs configured for a particular set of 16 pixel values is a *configuration context*. When the system changes the contents of the KCM LUTs in the reconfiguring set and the reconfiguring set switches to computation mode and the active set switches to reconfiguration mode, then we get a new configuration context and say that the computation subsystem undergoes a *configuration context switch*. It is important to realize that both active and reconfiguring sets reside simultaneously within the computation subsystem, and the computation subsystem undergoes a configuration context switch after every 16 clock cycles (refer to Section 2.2.2) to acquire a new configuration context.

The input data, that is, the filtering window coefficients, are routed to the modules as a set of sixteen inputs every clock cycle, one for each module. A set of sixteen pixel values (one per module) is input at each configuration context switch as configuration data, that is, data to be configured within the KCMs during their reconfiguration phase.

Passing data from one configuration context to another is required because we need to pass partial results of computations involving filtering windows that overlap pixels in two configuration contexts. Information generated by the last two modules in the computation subsystem, which initiate these computations, must reach the first two modules of the computation subsystem (now working in the next configuration context), which complete the computations (explained in Section 3.3). To pass this data, the computation subsystem maintains an array of six registers called *configuration start registers* connected to a 6×1 multiplexer called the *configuration start mux*. There is also a 1×6 demultiplexer called the *configuration end demux* present at the end of the computation subsystem. Configuration start registers receive data from module 15 in the computation subsystem through the configuration end demux. The configuration start mux passes the data stored in configuration start registers to module 0 of the computation subsystem. The configuration start mux and the configuration end demux both receive their select signals from the step counter. Figure 3.3 illustrates connections between first and last modules.

3.2 Working of a 1×3 Size Filtering Window

We have already described the essential details of our implementation, that is, the computation subsystem, which is enough for us to now describe the working of a 1×3 size filtering window on a 256×256 size image. Although this thesis deals with a 3×3 size filtering window, we first discuss a relatively simple case to convey the underlying thought in the implementation. Figure 3.4 shows three 1×3 windows overlapping the pixel at position [7,15] in an image.

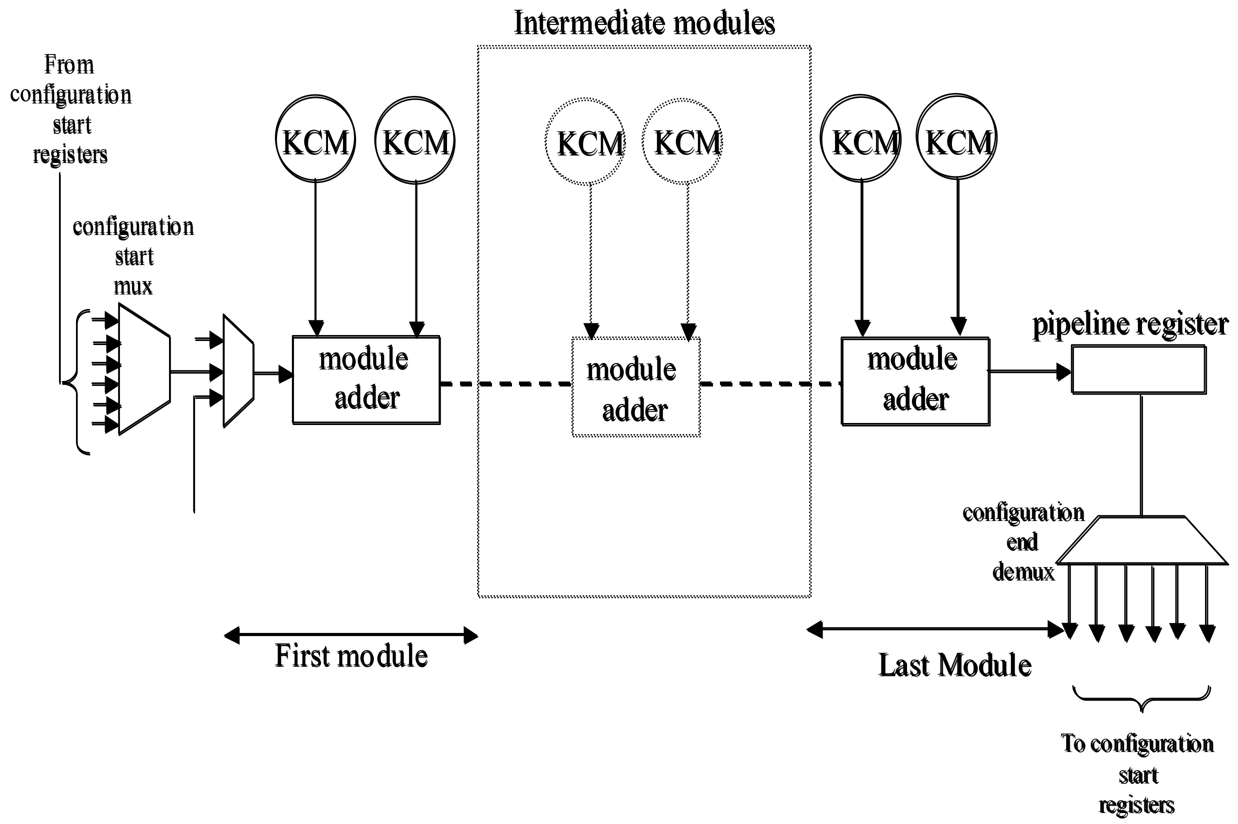

Figure 3.3 - First and last modules with configuration start mux and configuration end demux

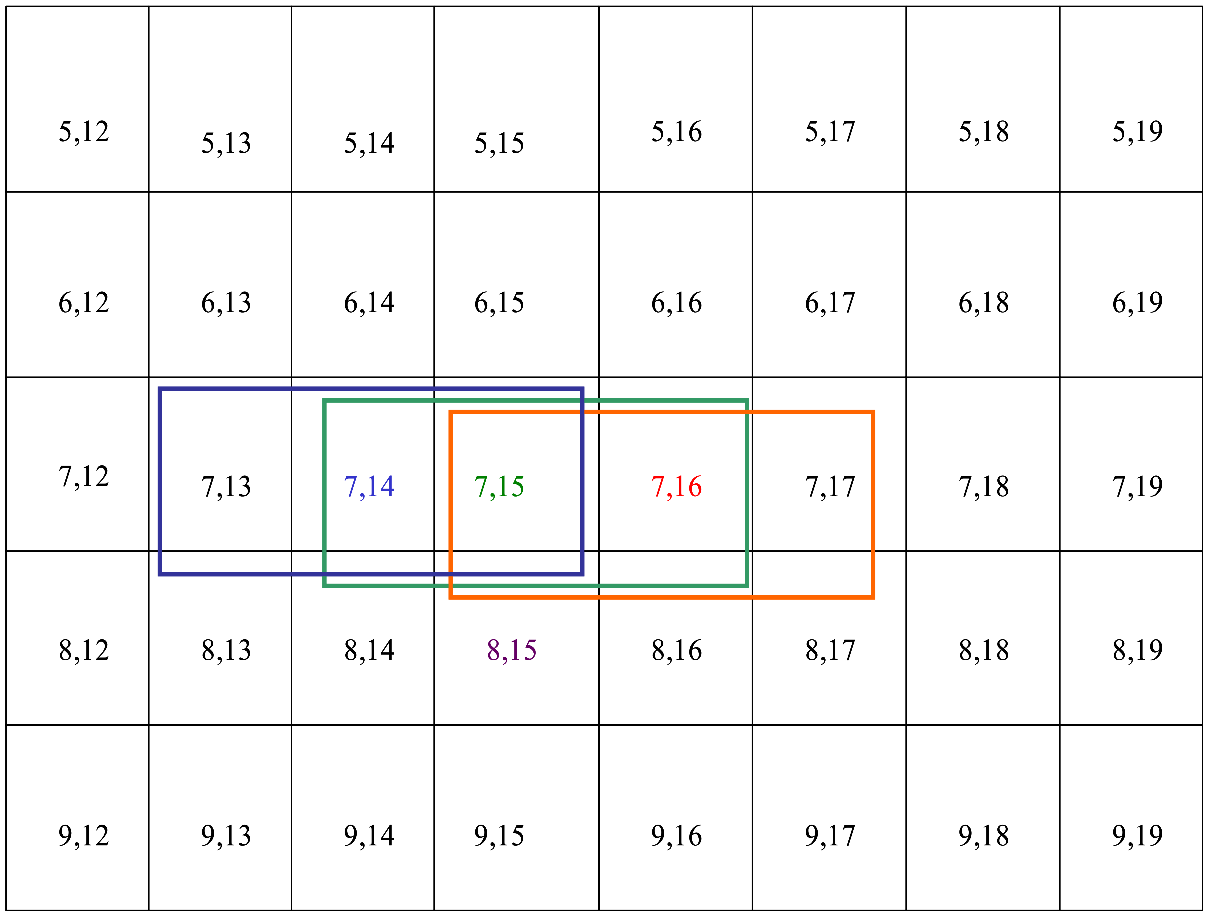

Figure 3.4 – Three 1×3 size filtering windows that overlap the pixel at position [7,15] in an image

Let us first provide some assumptions and notations.

- The image size is 256×256.
- Let $p[i,j]$ denote a pixel in an image and $v[i,j]$ its value, where i is the row number, j is the column number, and $0 \leq i, j \leq 255$.
- Let $nv[i,j]$ denote the new value of pixel $p[i,j]$, that is, its value after filtering.
- Let $cv[i,j,h]$ represent the value of a filtering window coefficient at position h of the window centered on pixel $p[i,j]$, where $-1 \leq h \leq 1$.
- Let $pd[i,j,h]$ denote the product $v[i,j+h] * cv[i,j,h]$.

- Each configuration context has three computation steps.
- Each computation step completes in one clock cycle.

The following equation shows the computations involved in applying a filtering window to generate new value *nv*[*i*,*j*] for pixel *p*[*i*,*j*].

$$nv[i,j] = \sum_{h=-1}^{1} v[i,j+h]*cv[i,j,h]$$

$$= \sum_{h=-1}^{1} pd[i,j,h] \qquad (3.1)$$

Thus we can see that the computation of the new value of any pixel needs three multiplication and two addition operations. Our design realizes this in three steps of computation within three adjacent modules. KCMs perform the multiplication operations while module adders perform the additions. We will look at Equation 3.1 from two vantage points; we first describe it from the point of view of a module and then from the point of view of the computations involved in producing *nv*[*i*,*j*].

A module receives two inputs, the filtering window coefficient (*KCM input*) and the data from its previous module (*adder input*). The KCM generates the product of the filtering window coefficient and the pixel value (configured data) called the *KCM output* and feeds this to the module adder which sums the KCM output with the adder input to produce the *module output*. Each module within the computation subsystem works in a similar fashion.

We discuss below the first vantage point, the computations performed by one module. Figure 3.5 gives the pseudocode for filtering a 256×256 size image using a 1×3 size filtering window. We now explain the computations performed by one module configured with *v*[*i*,*j*] within procedure *One_Dim*.

Step 0: The module receives *cv*[*i*,*j*+1,-1] as KCM input and zero as adder input from its zero register. The module adder sums KCM output *pd*[*i*,*j*+1,-1] to the adder input to produce *pd*[*i*,*j*+1,-1] as module output and passes this to the next module.

Step 1: The module receives a value *cv*[*i*,*j*,0] as KCM input and the module output of its previous module *pd*[*i*,*j*,-1] (produced in Step 0) as adder input. The module adder sums KCM output *pd*[*i*,*j*,0] to the adder input to produce *pd*[*i*,*j*,-1] + *pd*[*i*,*j*,0] as the module output and passes this to the next module.

Step 2: The module receives a value *cv*[*i*,*j*-1,1] as KCM input and the module output of its previous module *pd*[*i*,*j*-1,-1] + *pd*[*i*,*j*-1,0] (produced in Step 1) as adder input. The module adder sums KCM output *pd*[*i*,*j*-1,1] to the adder input to produce *nv*[*i*,*j*-1] and passes this to the I/O pins.

```
for i ← 0 to 255
        for k ← 0 to 255 in steps of 16
                for all j, where k ≤ j ≤ k+15
                        r = j mod 16  /* The KCM of module r  has  v[i,j] as constant */
                         in ← 0;
                        out ← I/O pins;

Procedure               Step 0: Adder( r ) ← KCM( r ) + in;
One_Dim(in,out)         Step 1: Adder( r ) ← KCM( r ) + Adder( r-1 );
                        Step 2: Adder( r ) ← KCM( r ) + Adder( r-1 );
                        out ← Adder( r );
```

Figure 3.5 – Pseudocode for filtering a 256×256 image using a 1×3 size filtering window

We now describe the second vantage point, that is, the generation of the new value of one pixel *p*[*i*,*j*]. Below is the description of the three computation steps.

Step 0: A module configured with *v*[*i*,*j*-1] initiates production of *nv*[*i*,*j*]. It receives *cv*[*i*,*j*,-1] as KCM input and zero as adder input. The KCM produces *pd*[*i*,*j*,-1] as the KCM output, which the module adder sums with the adder input to produce *pd*[*i*,*j*,-1] as the module output, and passes this to the next module.

Step 1: The next module, configured with *v*[*i*,*j*], receives *cv*[*i*,*j*,0] as KCM input and *pd*[*i*,*j*,-1] as the adder input (produced in Step 0). The KCM produces *pd*[*i*,*j*,0] as the KCM output which the module adder sums with the adder input to produce *pd*[*i*,*j*,-1] + *pd*[*i*,*j*,0] as the module output and passes this to the next module.

Step 2: The next module, configured with *v*[*i*,*j*+1], receives *cv*[*i*,*j*,1] as the KCM input and *pd*[*i*,*j*,-1] + *pd*[*i*,*j*,0] as the adder input (produced in Step 1). The KCM produces *pd*[*i*,*j*,1] as the KCM output, which the module adder sums with the adder input to produce *nv*[*i*,*j*] as the module output and passes this to the I/O pins.

To calculate the new value of pixel *p*[7,15] shown in Figure 3.4, we can put values in Equation 3.1 to deliver

$$
\begin{aligned}
nv[7,15] &= \sum_{h=-1}^{1} v[7,15+h]*cv[7,15,h] \\
&= pd[7,15,-1] + pd[7,15,0] + pd[7,15,1].
\end{aligned}
$$

Figure 3.6 illustrates the three computation steps in which a module configured with *v*[7,15] is involved and the three computation steps that produce *nv*[7,15]. The figure shows a module in each column while three rows correspond to the three computation steps as defined in procedure *One_Dim*. The KCMs are configured with the values shown at the top. The arrows show the three computation flows to which the module configured for *v*[7,15] contributes. For these flows, the figure shows the adder inputs received by the modules in the three steps. The computation for *nv*[7,15] starts at the module configured with *v*[7,14] and ends at the module configured with *v*[7,16], which sends *nv*[7,15] to the I/O pins. KCM inputs to modules are not shown. Values depicted as *pd* denote the KCM outputs. The values shown in dashed boxes at the bottom are new values for the pixels. The arrows depict the flow of data and also signify an addition operation. The tail depicts the source module and the head the destination module. Moving along arrows represents an addition of the adder input from the source and the KCM output of the destination module.

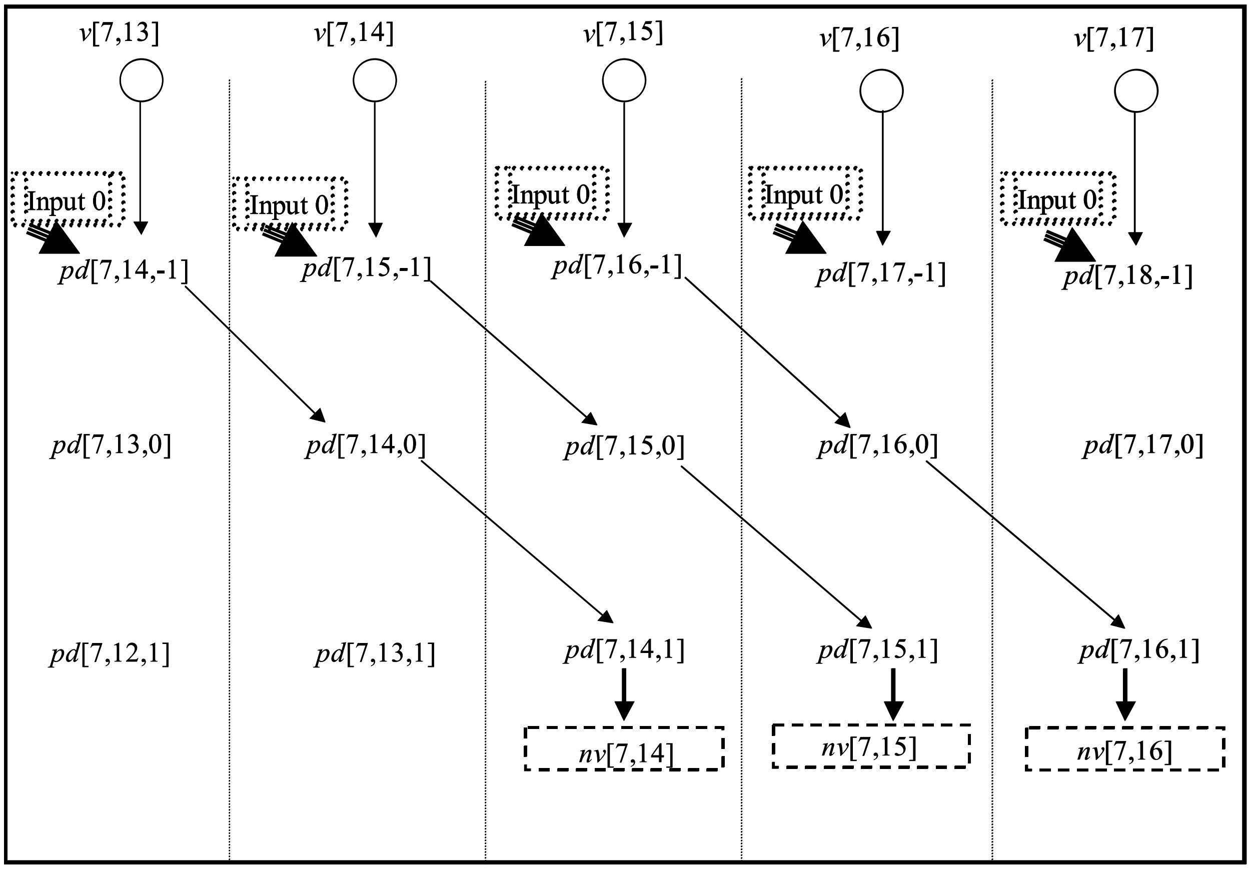

Figure 3.6 - Production of new values of pixels *p*[7,14], *p*[7,15], and *p*[7,16]

3.3 Working of a 3×3 Size Filtering Window

We are now ready to present the working of a 3×3 size filtering window by extending the less complex case of a 1×3 size filtering window. We will make use of the procedure *One_Dim* as defined in Figure 3.5 to develop pseudocode for the case of a 3×3 size filtering window. Figure 3.7 shows the nine 3×3 size windows that overlap the pixel at position *p*[7,15] in an image.

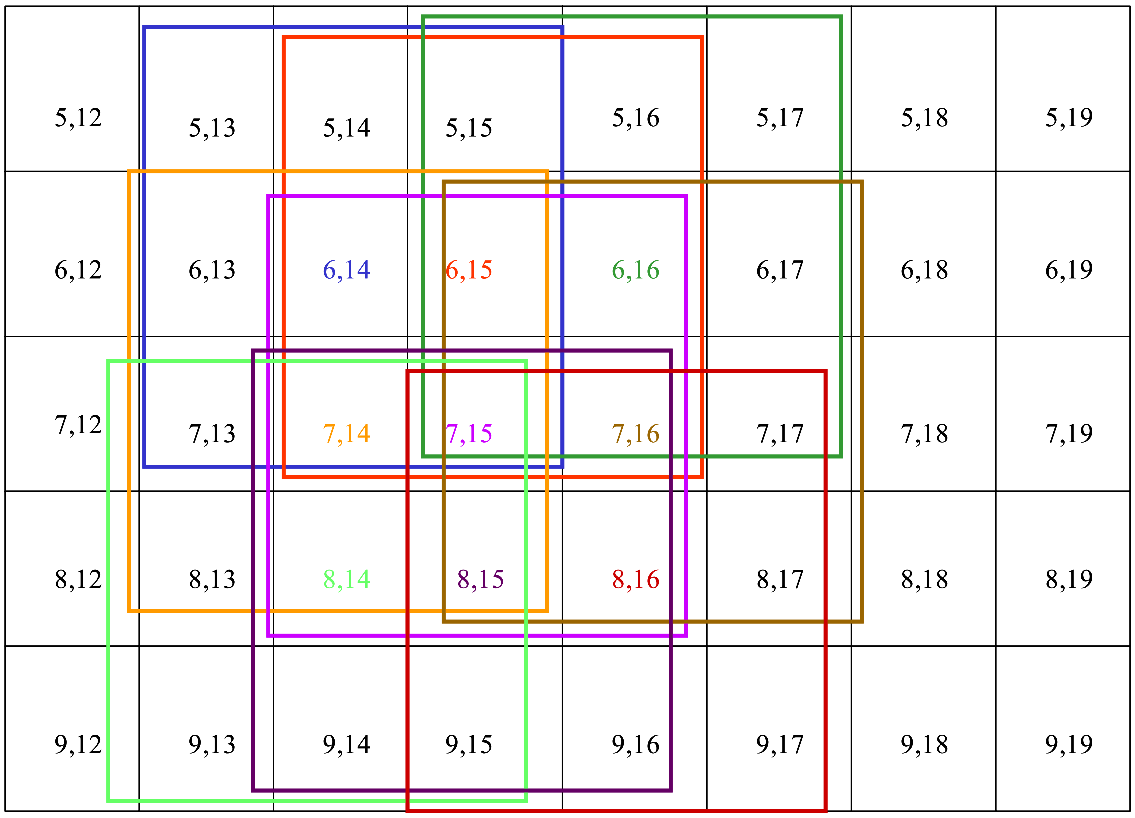

Figure 3.7 - Nine 3×3 size filtering windows overlapping the pixel at position *p*[7,15]

3.3.1 Concept of Window-Row Sums

To extend the case of a 1×3 size filtering window to a 3×3 size filtering window, we introduce here the concept of *window-row sum*. It is very much the underlying idea behind the entire implementation. For a 1×3 size filtering window, computing the new value of a pixel required three steps and the filtering window spanned only one row of the image, so there was no need to know pixel values from other rows of the image. Since a 3×3 size window spans three rows of the image, we need pixel values from three adjacent rows. As mentioned earlier, we receive the pixel values of the image in row major order; hence, we can process only one row of an image at a time. Our approach is to consider the 3×3 size filtering window as three separate but adjacent 1×3 size filtering windows stacked directly on top of each other and produce output as we did earlier in Section 3.2. The difference here being that the results output by the three different 1×3 size filtering windows must be added together to produce the new value of a pixel filtered by a 3×3 size filtering window. Call the output produced by each 1×3 size filtering window as a window-row sum. We use the block memories as described in Section 3.6 to store the window-row sums or sums of window-row sums.

We can look at the working of a 3×3 size filtering window from two vantage points. Section 3.3.2 discusses the first, that is, data flow within one module (similar to as described for a 1×3 size window). We discuss in Section 3.3.3 the second vantage point, that is, production of a new value of one pixel involving movement of data among adjacent modules including production and addition of window-row sums. Of course, we access block memory to store and retrieve the window-row sums.

3.3.2 Working of a Module

We must provide some more assumptions and notations in addition to those of Section 3.2.

- The range of gray levels is 0 to 255 (can be represented by an 8-bit binary number).
- The filtering window size is 3×3.
- The number of modules in the computation subsystem = 16.

- Window coefficients are in 8-bit fixed-point format. (Typically the filtering window coefficients are less than unity.)
- Since the filtering window is adaptive, coefficient values of a filtering window centered at one position may be different from those of a window centered at a different position.
- Call the top row of the 3×3 size filtering window as the *top window row* and the window-row sum produced by it is as the *top window-row sum.* Define *middle window row, middle window-row sum, bottom window row,* and *bottom window-row sum* similarly.
- Let $[g,h]$ denote the position of a coefficient within a filtering window, where $-1 \leq g,h \leq 1$ and [0,0] is the center position. Let $cv[i,j,g,h]$ represent the value of the filtering window coefficient at position $[g,h]$ of the window centered on pixel $p[i,j]$.
- Let $rs(g)[i,j]$ denote the window-row sum produced for row g of the filtering window centered at $p[i,j]$.
- Let $pd[i,j,g,h]$ denote $v[i+g,j+h]*cv[i,j,g,h]$.

To compute new value of a pixel $p[i,j]$, we need to compute the following equation.

$$nv[i,j] = \sum_{g=-1}^{1} \sum_{h=-1}^{1} v[i+g,j+h]*cv[i,j,g,h] \tag{3.2}$$

Removing the outer summation in Equation 3.2 breaks the equation into three window-row sums.

$$rs(-1)[i,j] = \sum_{h=-1}^{1} pd[i,j,-1,h] \tag{3.3}$$

$$rs(0)[i,j] = \sum_{h=-1}^{1} pd[i,j,0,h] \tag{3.4}$$

$$rs(1)[i,j] = \sum_{h=-1}^{1} pd[i,j,1,h] \tag{3.5}$$

Figure 3.8 shows the pseudocode for the working of a 3×3 size window using procedure *One_Dim*. Figure 3.9 shows the pseudocode upon expanding the procedure *One_Dim*.

```
for i← 0 to 255
        for k← 0 to 255 in steps of 16
                for all j, where k ≤ j ≤ k+15
                        r = j mod 16  /* The KCM of module r  has  v[i,j] as constant */
                        One_Dim(0, memory);            /* input 0 and output to memory*/
                        One_Dim(memory, memory);  /* input from memory, output to memory*/
                        One_Dim(memory, I/O pins);  /* input from memory, output to I/O pins*/
```

Figure 3.8 – Pseudocode for working of a 3×3 size filtering window using procedure *One-Dim*

The beauty of this implementation is regularity of data flow through the circuit. All modules receive their inputs at the same time, perform the same sequence of operations, and produce their outputs simultaneously. Table 3.1 provides information on the value produced by a module with KCM configured for *v*[*i*,*j*]. Each row of the table corresponds to a particular computation step of the pseudocode of Figure 3.9 and provides information about data flow within the module.

We describe below the second computation step (Step 1) that a module undergoes in a configuration context as an example. This will make it easier to interpret the contents of Table 3.1. Please recall that each module works in exactly the same way and the module KCM is configured for *v*[*i*,*j*].

Step 1: The KCM receives *cv*[*i*+1,*j*,-1,0] as its input and produces *pd*[*i*+1,*j*,-1,0] as its output. The module receives the module output of its previous module *pd*[*i*+1,*j*,-1,-1] (produced in Step

0) as adder input. The module adder sums the KCM output and the adder input to produce *pd*[*i*+1,*j*,-1,-1] + *pd*[*i*+1,*j*,-1,0] as the module output and passes this to the next module.

```
for $i \leftarrow$ 0 to 255
    for $k \leftarrow$ 0 to 255 in steps of 16
        for all $j$, where $k \le j \le k+15$
            $r = j$ mod 16  /* The KCM of module $r$ has $v[i,j]$ as constant */
            Step 0: Adder( $r$ ) ← KCM( $r$ ) + 0;

            Step 1: Adder( $r$ ) ← KCM( $r$ )+ Adder( $r$-1 );

            Step 2: Adder( $r$ ) ← KCM( $r$ ) + Adder( $r$-1 );
                    Memory ← Adder( $r$ );

            Step 3: Adder( $r$ ) ← KCM( $r$ ) + Memory;

            Step 4: Adder( $r$ ) ← KCM( $r$ ) + Adder( $r$-1 );

            Step 5: Adder( $r$ ) ← KCM( $r$ ) + Adder( $r$-1 );
                    Memory ← Adder( $r$ );

            Step 6: Adder( $r$ ) ← KCM( $r$ ) + Memory;

            Step 7: Adder( $r$ ) ← KCM( $r$ ) + Adder( $r$-1 );

            Step 8: Adder( $r$ ) ← KCM( $r$ ) + Adder( $r$-1 );
                    I/O pins ← Adder( $r$ );
```

Figure 3.9 – Pseudocode for working of a 3×3 size window upon expanding procedure *One_Dim*

Table 3.1 - Data flow within a module with KCM configured for *v*[*i*,*j*]

Step no.	KCM input	KCM output	Adder input	Source of adder input	Value produced by adder (module output)	Comments
0	*cv*[*i*+1,*j*+1,-1,-1]	*pd*[*i*+1,*j*+1,-1,-1]	0	zero register	*pd*[*i*+1,*j*+1,-1,-1] + 0	zero fed to adder
1	*cv*[*i*+1,*j*,-1,0]	*pd*[*i*+1,*j*,-1,0]	*pd*[*i*+1,*j*,-1,-1]	Previous module	*pd*[*i*+1,*j*,-1,-1]+ *pd*[*i*+1,*j*,-1,0]	
2	*cv*[*i*+1,*j*-1,-1,1]	*pd*[*i*+1,*j*-1,-1,1]	*pd*[*i*+1,*j*-1,-1,-1]+ *pd*[*i*+1,*j*-1,-1,0]	Previous module	*rs*(-1)[*i*+1,*j*-1]	output to mem
3	*cv*[*i*,*j*+1,0,-1]	*pd*[*i*,*j*+1,0,-1]	*rs*(-1)[*i*,*j*+1]	Block memory	*rs*(-1)[*i*,*j*+1]+ *pd*[*i*,*j*+1,0,-1]	read from mem
4	*cv*[*i*,*j*,0,0]	*pd*[*i*,*j*,0,0]	*rs*(-1)[*i*,*j*]+ *pd*[*i*,*j*,0,-1]	Previous module	*rs*(-1)[*i*,*j*]+ *pd*[*i*,*j*,0,-1]+ *pd*[*i*,*j*,0,0]	
5	*cv*[*i*,*j*-1,0,1]	*pd*[*i*,*j*-1,0,1]	*rs*(-1)[*i*,*j*-1]+ *pd*[*i*,*j*-1,0,-1]+ *pd*[*i*,*j*-1,0,0]	Previous module	*rs*(-1)[*i*,*j*-1]+ *rs*(0)[*i*,*j*-1]	output to mem
6	*cv*[*i*-,*j*+1,1,-1]	*pd*[*i*-1,*j*+1,1,-1]	*rs*(-1)[*i*-1,*j*+1]+ *rs*(0)[*i*-1,*j*+1]	Block memory	*rs*(-1)[*i*-1,*j*+1]+ *rs*(0)[*i*-1,*j*+1]+ *pd*[*i*-1,*j*+1,1,-1]	read from mem
7	*cv*[*i*-1,*j*,1,0]	*pd*[*i*-1,*j*,1,0]	*rs*(-1)[*i*-1,*j*]+ *rs*(0)[*i*-1,*j*]+ *pd*[*i*-1,*j*,1,-1]	Previous module	*rs*(-1)[*i*-1,*j*]+ *rs*(0)[*i*-1,*j*]+ *pd*[*i*-1,*j*,1,-1]+ *pd*[*i*-1,*j*,1,0]	
8	*cv*[*i*-1,*j*-1,1,1]	*pd*[*i*-1,*j*-1,1,1]	*rs*(-1)[*i*-1,*j*-1]+ *rs*(0)[*i*-1,*j*-1]+ *pd*[*i*-1,*j*-1,1,-]+ *pd*[*i*-1,*j*-1,1,0]	Previous module	nv[*i*-1,*j*-1]	to I/O pins

Figure 3.11 illustrates these nine steps using the specific case of a module configured for *v*[7,15]. Figure 3.7 illustrates the nine filtering windows that overlap pixel *p*[7,15]. We provide in Figure 3.10 some arrow notations that are helpful in understanding Figures 3.11-3.14, where *pd* values are KCM output values and match the 'KCM output' column of Table 3.1.

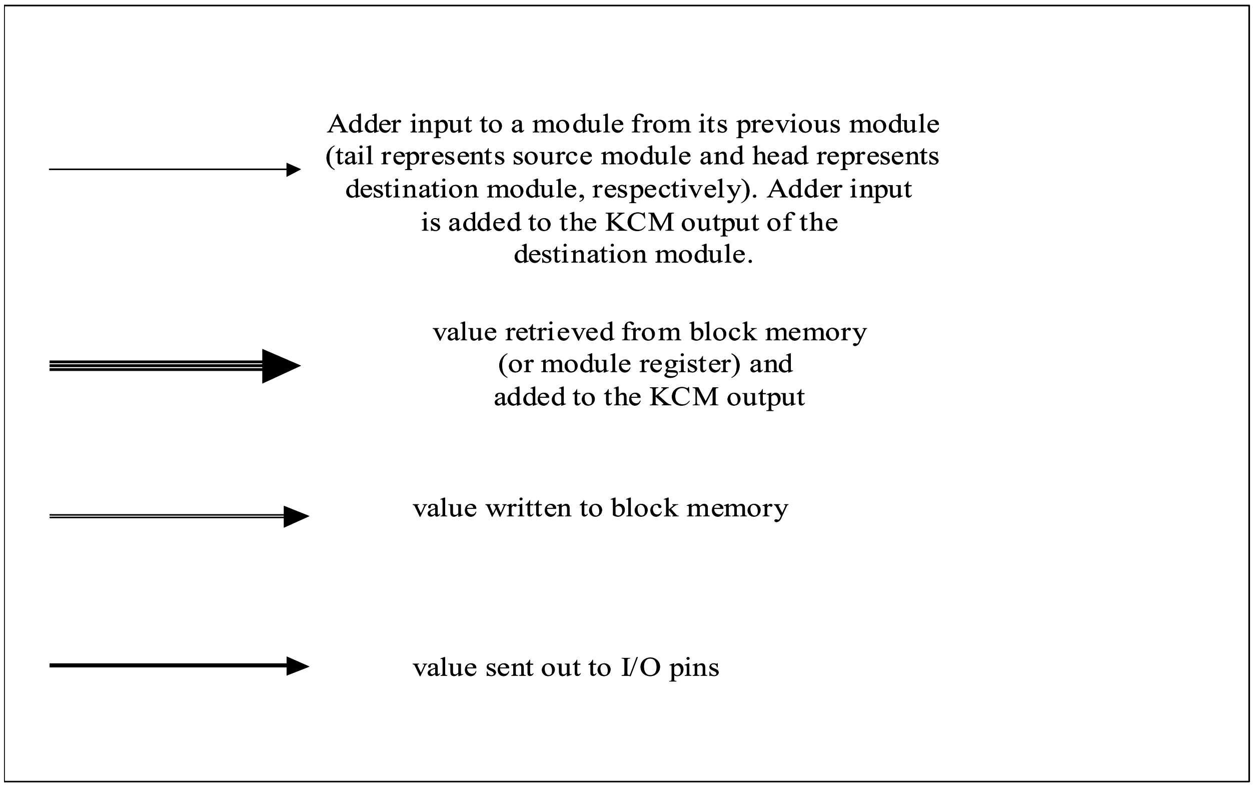
Adder input to a module from its previous module (tail represents source module and head represents destination module, respectively). Adder input is added to the KCM output of the destination module.
value retrieved from block memory (or module register) and added to the KCM output
value written to block memory
value sent out to I/O pins

Figure 3.10 – Arrow notations

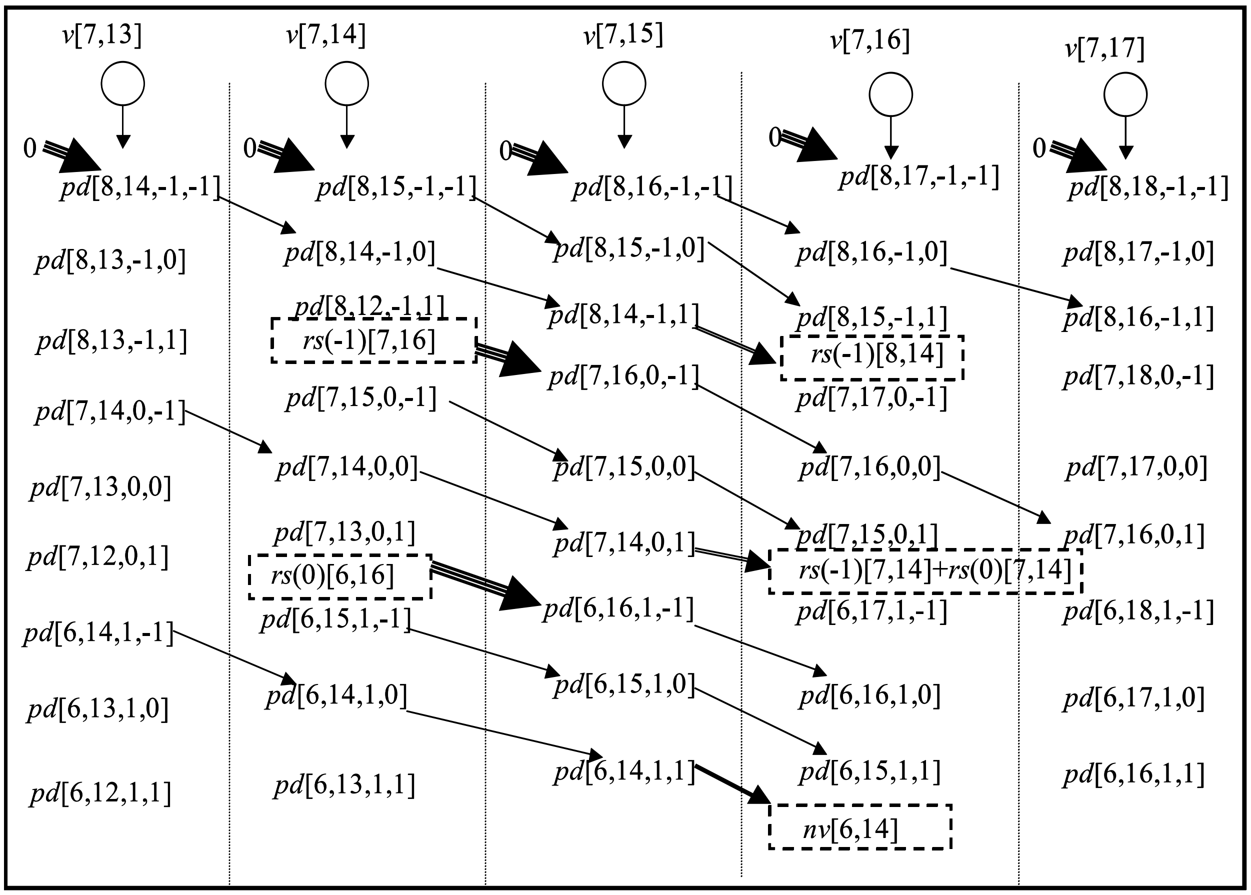

Figure 3.11 - Nine computation steps of a module configured for *v*[7,15]

Let us explain the contents of Figure 3.11. The five columns show five adjacent modules with KCMs configured with values as shown on top. The values enclosed in boxes are values read from or written to memory or values passed to I/O pins. Note that the five modules are configured for the same pixel values as the modules in Figure 3.6. Figure 3.6 illustrates the computations performed by a module for the case of a 1×3 size filtering window, while Figure 3.11 illustrates the same in the case of a 3×3 size filtering window. Of course, modules in the case of a 1×3 size filtering window do not perform any memory access operations. The modules in Figure 3.6 produce the new value of a pixel in Step 2, while modules in Figure 3.11 produce the new value of a pixel in Step 8.

Note in Figure 3.11 that the module configured for *v*[7,15] (third column) initiates top, middle, and bottom window-rows sums for pixels *p*[8,16], *p*[7,16], and *p*[6,16] at Steps 0, 3, and 6, respectively. At Steps 1, 4, and 7, it contributes to the window-row sums for *p*[8,15], *p*[7,15], and *p*[6,15], respectively. It completes top, middle, and bottom window-rows sums for pixels *p*[8,14], *p*[7,14], and *p*[6,14] at Steps 2, 5, and 8, respectively. This is how a module in one configuration context works for nine computation steps, contributing towards the new value of a different pixel at each computation step, before reconfiguring its KCM. A look at Figure 3.7 would confirm that pixel *p*[7,15] indeed occupies a position within the nine filtering windows centered on the above mentioned pixels.

3.3.3 Production of a New Value of a Pixel

As promised earlier, this section explains the working of 3×3 size filtering window from the second vantage point, that is, computations involving production of the new value of a pixel, and establishes the correctness of the filtering computation.

Figure 3.7 shows that the window centered on pixel *p*[7,15] overlaps nine different pixels *p*[6,14], *p*[6,15], *p*[6,16], *p*[7,14], *p*[7,15], *p*[7,16], *p*[8,14], *p*[8,15], and *p*[8,16]. Pixel values of all these nine pixels contribute towards *nv*[7,15].

From the point of view of computations for the new value for pixel *p*[*i*,*j*], three modules will participate: modules (*j*-1) modulo 16, *j* modulo 16, and (*j*+1) modulo 16. Let *r*-1, *r*, and *r*+1 denote these module indices, respectively. When processing row *i*-1 of the image, module *r*-1 configured with value *v*[*i*-1,*j*-1] initiates production of the top window-row sum, and module *r*+1 configured with value *v*[*i*-1,*j*+1] completes it to produce *rs*(-1)[*i*,*j*] and store it in block memory. Similarly, when processing row *i* of the image, module *r*-1 configured with value *v*[*i*,*j*-1] reads *rs*(-1)[*i*,*j*] from block memory, adds it to its KCM output, and initiates production of the middle window-row sum. Module r+1 configured with value $v[i,j+1]$ completes it to produce the value $rs(-1)[i,j] + rs(0)[i,j]$ and writes it to block memory. Finally, when processing row i+1 of the image, module r-1 configured with $v[i+1,j-1]$ reads value $rs(-1)[i,j] + rs(0)[i,j]$ from block memory, adds it to its KCM output, and initiates production of the bottom window-row sum. Module r+1 configured with value $v[i+1,j+1]$ completes it to produce $nv[i,j]$.

We seek to elaborate more on this by providing a suitable example. We again use Figure 3.7 and trace the steps that produce the new value of pixel $p[7,15]$. Do note that production of $nv[7,15]$ needs nine different pixel values that are multiplied with the nine coefficients of the filtering window centered on $p[7,15]$. The nine pixel values required in this case are $v[6,14]$, $v[6,15]$, and $v[6,16]$ all in row six of the image covered by the top window-row; $v[7,14]$, $v[7,15]$, and $v[7,16]$ all in row seven of the image covered by the middle window-row; and $v[8,14]$, $v[8,15]$, and $v[8,16]$ all in row eight of the image covered by the bottom window-row.

View a 3×3 filtering window as three 1×3 filtering windows working at different times. Since only one row of the image can be processed in a configuration context, the three 1×3 size filtering windows work in three different configuration contexts. Figures 3.12-3.14 trace data flow within a particular portion of three different configuration contexts involved in the production of $nv[7,15]$. Values shown in dashed boxes are values written to or read from memory, or for the final result $nv[7,15]$, sent to the I/O pins. Please note that in each configuration context, the same numbered modules participate in producing $nv[7,15]$, that is, if modules 2, 3, and 4 participated in producing the top window-row sum for a certain pixel, then they will participate again later in other configuration contexts for the middle and bottom window-row sums for the same pixel.

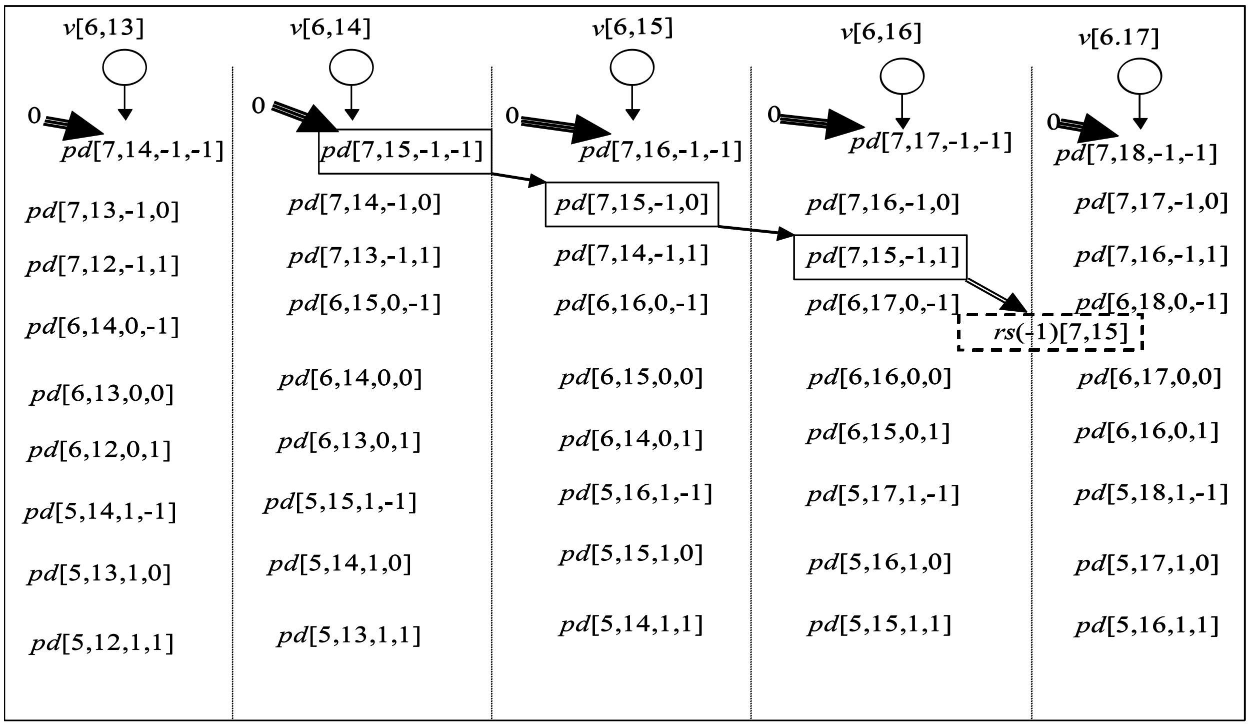

Figure 3.12 - Production of top window-row sum for *p*[7,15]

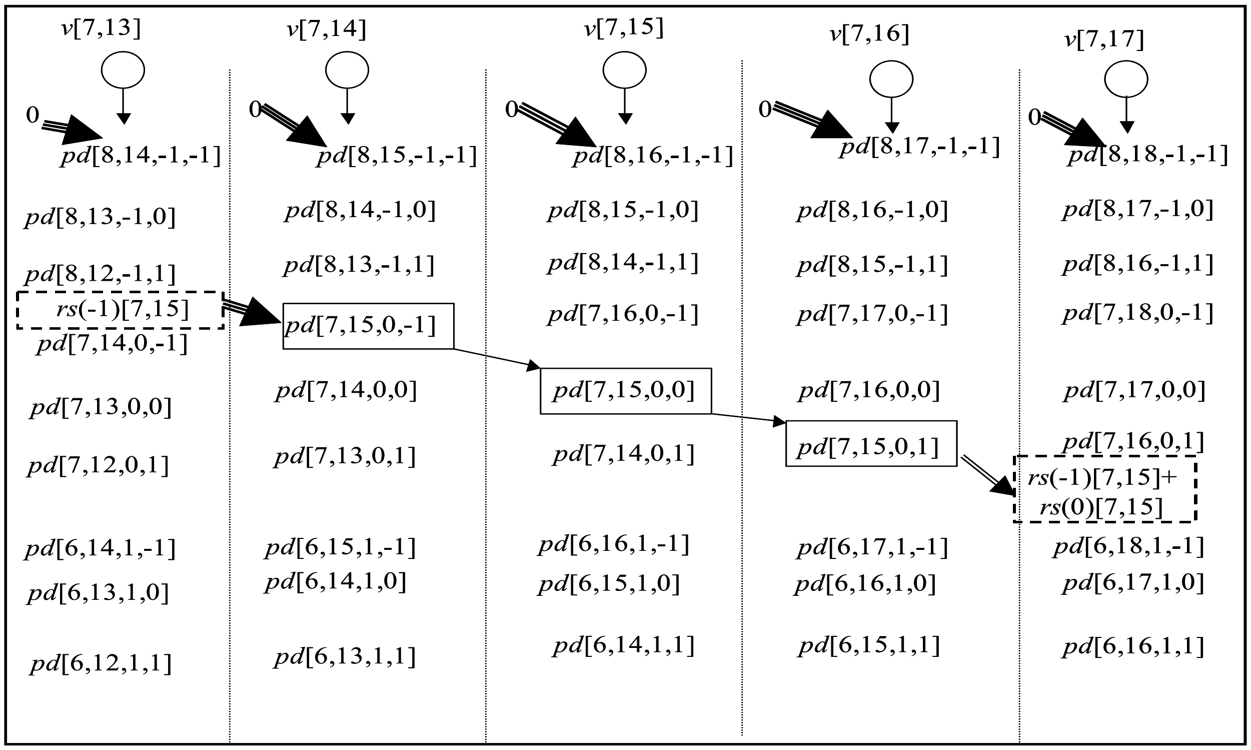

Figure 3.13 - Production of sum of top and middle window-sums for *p*[7,15]

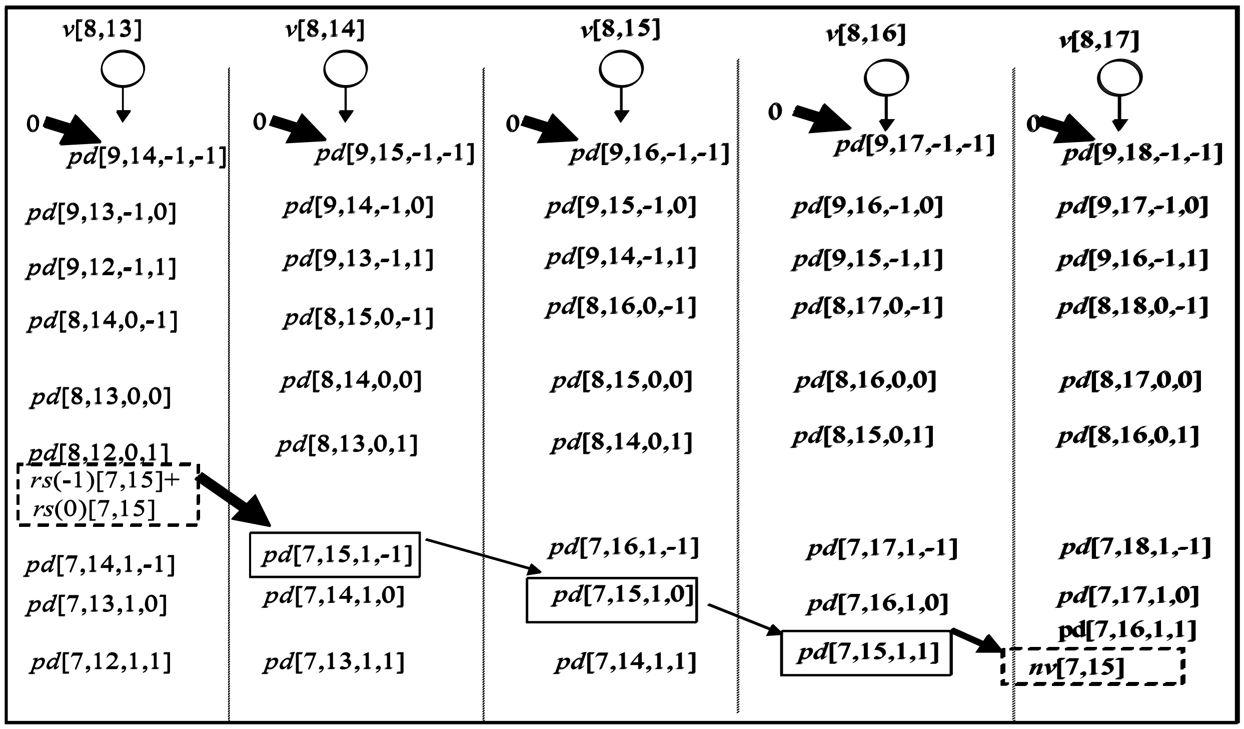

Figure 3.14 – New pixel value of pixel *p*[7,15] produced

3.3.4 Memory Access Pattern

In this section, we focus on the pattern of block memory access. Each module reads from one memory block and writes to another. Each memory block is read from one module and written to by another module. Specifically, each module configured for *v*[*i*,*j*] reads from block memory (*j*+2) modulo 16 when it initiates computation of middle or bottom window row-sums and writes to block memory *j* modulo 16 when it completes top or middle window row-sums.

Every module (except modules 14 and 15) reads from the same address in the block memories at Steps 3 and 6 during a configuration context. Similarly, every module writes to the same address during Steps 2 and 5. Thus, not only the data flow within a module but their memory access pattern also is regular in nature. Table 3.2 provides information about the block memories and their addresses accessed by a module configured with *v*[*i*,*j*] for its block memory read and write operations.

Table 3.2 Memory access pattern for a module configured for *v*[*i*,*j*]

Step no.	**Destination block memory of write**	**Destination address of write**	**Source block memory of read**	**Source address of read**
0	-	-	-	-
1	-	-	-	-
2	j modulo 16	$2*\lfloor j/16 \rfloor$ for even i $2*\lfloor j/16 \rfloor+32$ for odd i	-	-
3	-	-	(j+2) modulo 16	$2*\lfloor j/16 \rfloor$ for odd i $2*\lfloor j/16 \rfloor+32$ for even i
4	-	-	-	-
5	j modulo 16	$2*\lfloor j/16 \rfloor+1$ for even i $2*\lfloor j/16 \rfloor+33$ for odd i	-	-
6	-	-	(j+2) modulo 16	$2*\lfloor j/16 \rfloor+1$ for odd i $2*\lfloor j/16 \rfloor+33$ for even i
7	-	-	-	-
8	-	-	-	-

Table 3.1 shows that a module writes to block memory at Steps 2 and 5 while it reads block memory at Steps 3 and 6. Note that modules at Steps 2 and 5 store window-row sums belonging to new values of pixels in two adjacent rows of the image. Every one of 32 values written by module (j+2) modulo 16 when handling row i is read by module j modulo 16 when it handles row i+1. Because of the write/read sequence, one module writes 33 values before another one reads the first value. Although 33 addresses in each block memory are sufficient, we

choose to have 64 addresses as it makes handling of memory accesses easier. Since Block SelectRAMs available on Xilinx Virtex™-E FPGAs are only 16 bits wide, we take only the 16 most significant bits of the module output and store them in block memories. Since the filtering window coefficients are fixed point numbers, we do not lose precision in truncating the module output to 16 bits as the new value of a pixel is only 8 bits wide.

A simple but suitable example (Figure 3.15) will help to explain the memory access pattern. The example includes only four modules numbered 0 through 3, and four block memories numbered 0 through 3. The rectangles enclosed by dark lines show the pixel groups whose values are configured into the four modules in a configuration context. Let us concentrate on the filtering window centered on pixel *p*[7,14]. When processing row six of the image, module 1 configured with *v*[6,13] initiates production of *rs*(-1)[7,14] which module 3 completes and writes to block memory 3. When row seven is processed, module 1 configured with *v*[7,13] reads *rs*(-1)[7,14] from block memory 3 and initiates production of *rs*(0)[7,14]. Module 3 completes it and writes *rs*(-1)[7,14] + *rs*(0)[7,14] to block memory 3. Similarly while processing row eight of the image, module 1 reads *rs*(-1)[7,14] + *rs*(0)[7,14] and initiates production of *rs*(1)[7,14] which module 3 completes to produce *nv*[7,14] and sends it out to the I/O pins.

As mentioned earlier, modules 14 and 15 access a different address when reading from block memories. This is because when these two modules initiate the middle and bottom window-row sums for a pixel, the window centered on that pixel spans across two configuration contexts. In general, the last (w-1) modules of the computation subsystem, where $w \times w$ is the filter size, will read addresses, that are different from the addresses that other modules read. This property of the memory access pattern depends only upon the size of the filtering window. Our next example illustrated in Figure 3.16, uses a 3×3 size filtering window to explain this phenomenon in memory access pattern. To keep the illustration sufficiently small it uses four modules and four block memories instead of 16 each. Modules 2 and 3 play the same role that modules 14 and 15 play in the 16 module case. In Figure 3.16 a window centered on *p*[7,16] spans across two configuration contexts, so module 3 configured with *v*[7,15] reads window-row sum *rs*(-1)[7,16] that module 1 configured with *v*[6,17] had completed and stored in block memory 1. As each module writes to block memory two times in one configuration context, the locations that modules 2 and 3 read have an address two higher than the locations that modules 0 and 1 read.

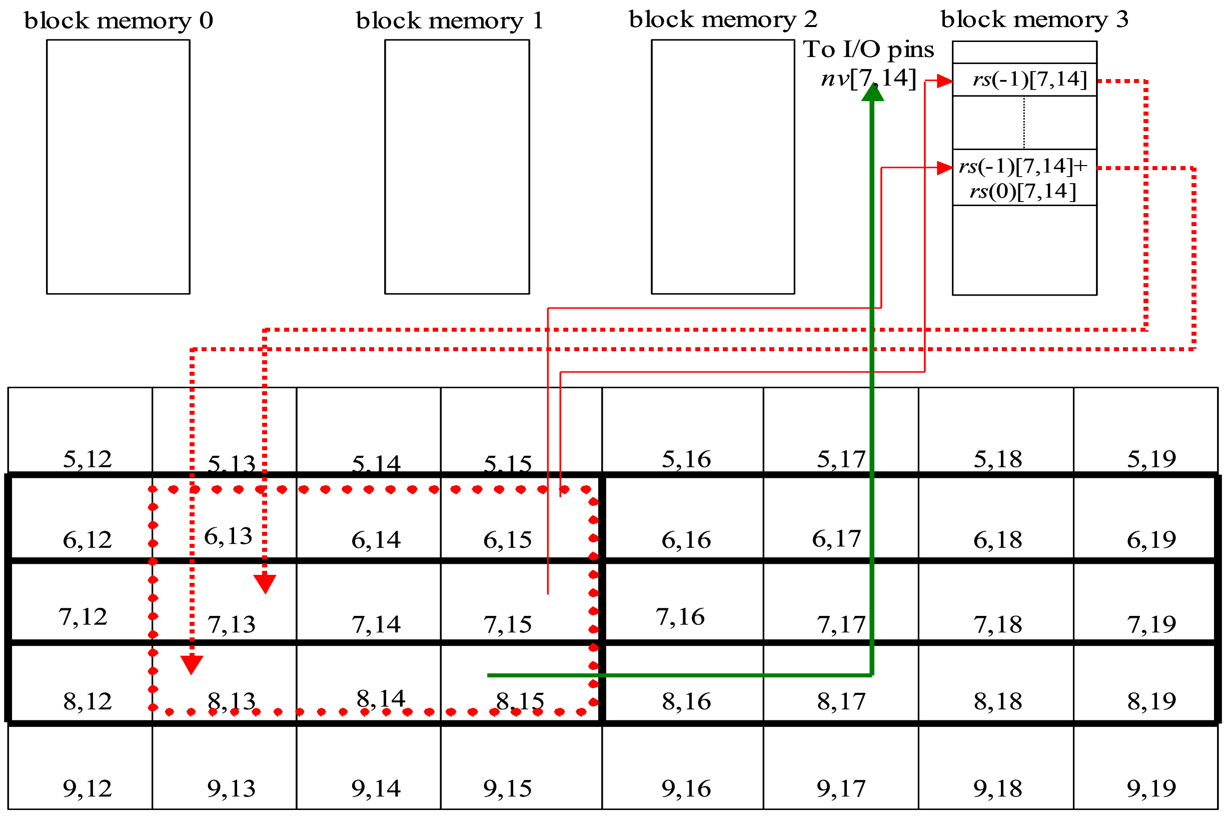

Figure 3.15 - Memory access pattern for window centered on *p*[7,14]

A similar logic applies to our implementation comprising 16 modules and 16 block memories. In the case of our implementation, modules 14 and 15 initiate window-row sums for filtering windows spanning across two configuration contexts. Thus, as explained above, the locations that modules 14 and 15 read have an address two higher than the locations that other modules in the computation subsystem read.

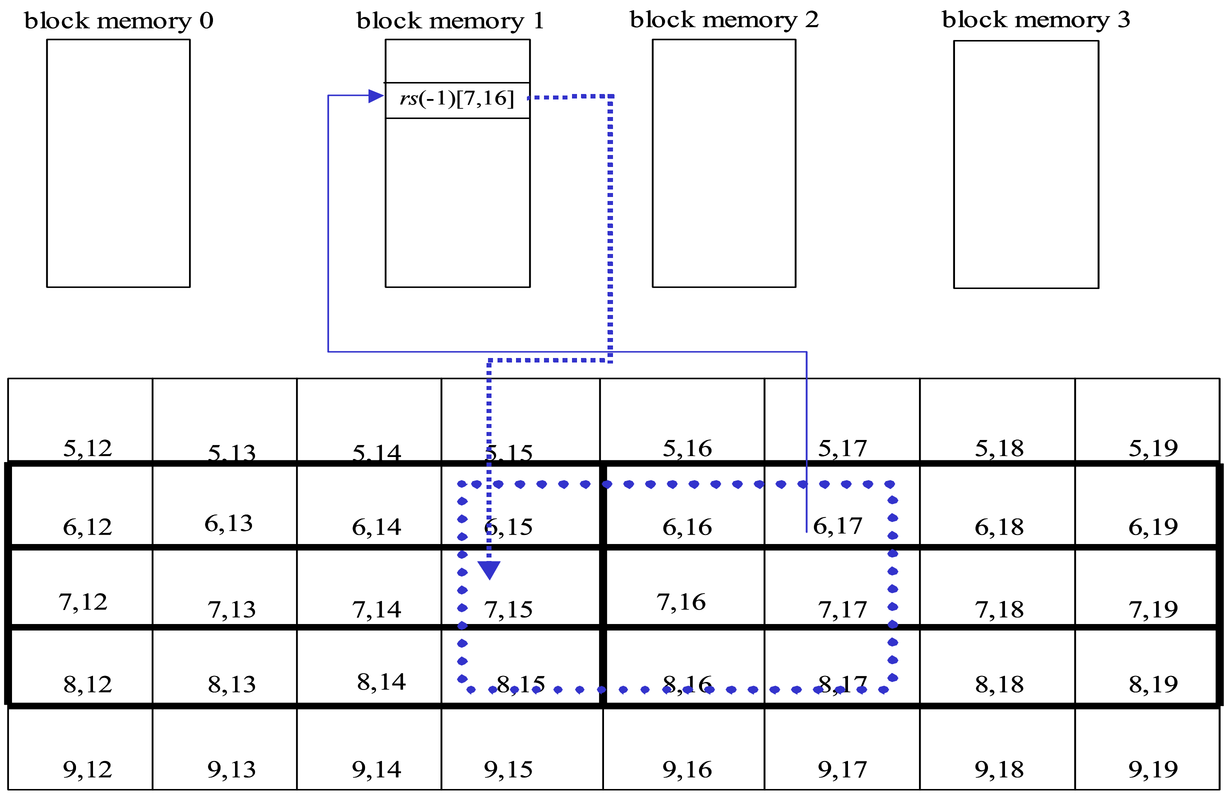

Figure 3.16 – Memory access pattern for window centered on *p*[7,16]

3.4 Using Run-Time Reconfiguration (RTR)

Earlier sections in this chapter concentrated on the working of modules and the data flow in them. This section focuses on how we utilize the concept of run-time reconfiguration to reduce the running time of the algorithm.

In the case of a spatially invariant filtering algorithm, the filtering window coefficients are fixed for the entire run of the algorithm, so we could configure the KCMs once with window coefficients and provide pixel values as input. Of course, this implementation would not need RTR.

In the case of the adaptive filtering algorithm, when the filtering window coefficients may have different values for filtering windows at different positions within the image, such an implementation is not feasible. An observation of data flow for an adaptive filter reveals that each pixel contributes towards the new pixel values of nine different pixels. Thus, we ascertain that a pixel value, once configured within a KCM as multiplier constant, can be used for nine clock cycles, a fact that cannot be guaranteed about window coefficients. In this way, we turned the implementation for a spatially invariant filter on its head to supply the window coefficients as input to KCMs and configure the KCMs with pixel values. This allowed us to calculate the time span for one configuration context and hence apply RTR effectively to shorten the running time of the adaptive filtering algorithm. Here we perform all computations that use a pixel value in nine consecutive computation steps, that is, within one configuration context, and then reconfigure the KCMs for a different pixel value. By using pixel values as configuration data for the KCMs, we reduce and regularize the number of reconfigurations required.

We have two KCMs per module so that we can configure one while the other one is active and in this way overlap time spent in reconfiguring the next configuration context with the computation time of the present configuration context. This is where we save time and shorten the running time of the algorithm. If we had used only one KCM per module, the computation and the reconfiguration phases could not be overlapped and the KCM would spend 16 cycles in reconfiguring itself on every configuration context switch. Having two KCMs thus serves to better utilize time. This approach is similar to the approach used by Wojko and ElGindy [WE] in implementing an FIR filter using RTR (refer to Section 2.5).

In an ideal situation when the time required to configure a configuration context is less than or the same as the time spent performing computation in a configuration context, we can reduce the running time by a high percentage. In the case of a 3×3 size filtering window, we have seven idle cycles on each configuration context switch as the computation phase takes nine clock cycles while the reconfiguration phase takes 16 clock cycles (refer to Section 2.2.1). If a 5×5 or larger size filtering window is used, then the entire reconfiguration phase for a KCM would be complete before a context switch is required as the computation phase would take 25 or more clock cycles. This is because a pixel would participate in computation of new pixel values of 25 or more pixels instead of nine.

3.5 I/O and Control Subsystem

The I/O and control subsystem described here builds on top of the computation subsystem described earlier. A host of controllers and other elements route the input data from the I/O pins of the FPGA to the modules and vice versa (Figure 3.17). The filter processes 16 configuration contexts for each row of the image. A four bit modulo up counter called the *configuration context counter* tracks the current configuration context. The *configuration context counter controller* (CCCC) updates the configuration context counter. A nine bit modulo up counter called *row counter* keeps track of the current row number of the image being processed. The *row counter controller* (RCC) updates the row counter. We have only one of each type of controller and counter for the entire design as all modules go through the same steps of computation.

We have 16 8-bit 1×2 demultiplexers called the *input demuxes* that route input data from the I/O pins of the FPGA to the active KCM of each module. We also have 16 8-bit 2×1 demultiplexers called the *pixel input demuxes* that route the pixel values to be configured into the reconfiguring KCM of each module. We have a set of 16 2×1 8-bit multiplexers called *output* *muxes* that route the final pixel value from the proper source, that is, either modules or boundary handling subsystem (refer to Section 3.7) to the I/O pins on the FPGA.

3.6 Memory Subsystem

We use 16 blocks of local memory (one per module) called *block memory*, each having 64 locations with each location 16 bits wide. We make use of BlockSelectRAMs of the Virtex™-E FPGA for this purpose. These block memories temporarily store partial results of the computation, and the modules write to and read from them.

Figure 3.18 illustrates the memory subsystem. The regular flow of data through the modules (refer to Section 3.3.2) obviates the need for a complicated address generation unit. Instead, we use two 6-bit modulo up counters called *memory write counter* and *memory read counter* to locate block memory read and write addresses, respectively. We initialize the memory write and read counters to zero and 32, respectively. Since all modules except module 14 and module 15 (explained in Section 3.3.4) in the computation subsystem read from the same location (from different block memories) at the same time, we need only two of each type of counter. We use memory registers to shorten the critical path of data so that the block memory speeds do not affect the performance of the design.

3.7 Image Boundary Handling Subsystem

The design includes a separate image boundary handling circuit that is devoted to the task of handling the values of pixels at the image boundary, that is, the top and bottom rows and the left and right columns of the image. These pixels contribute towards the new values of other pixels but windows centered on them extend beyond the image. Therefore they do not undergo filtering, and their original values are passed to the I/O pins.

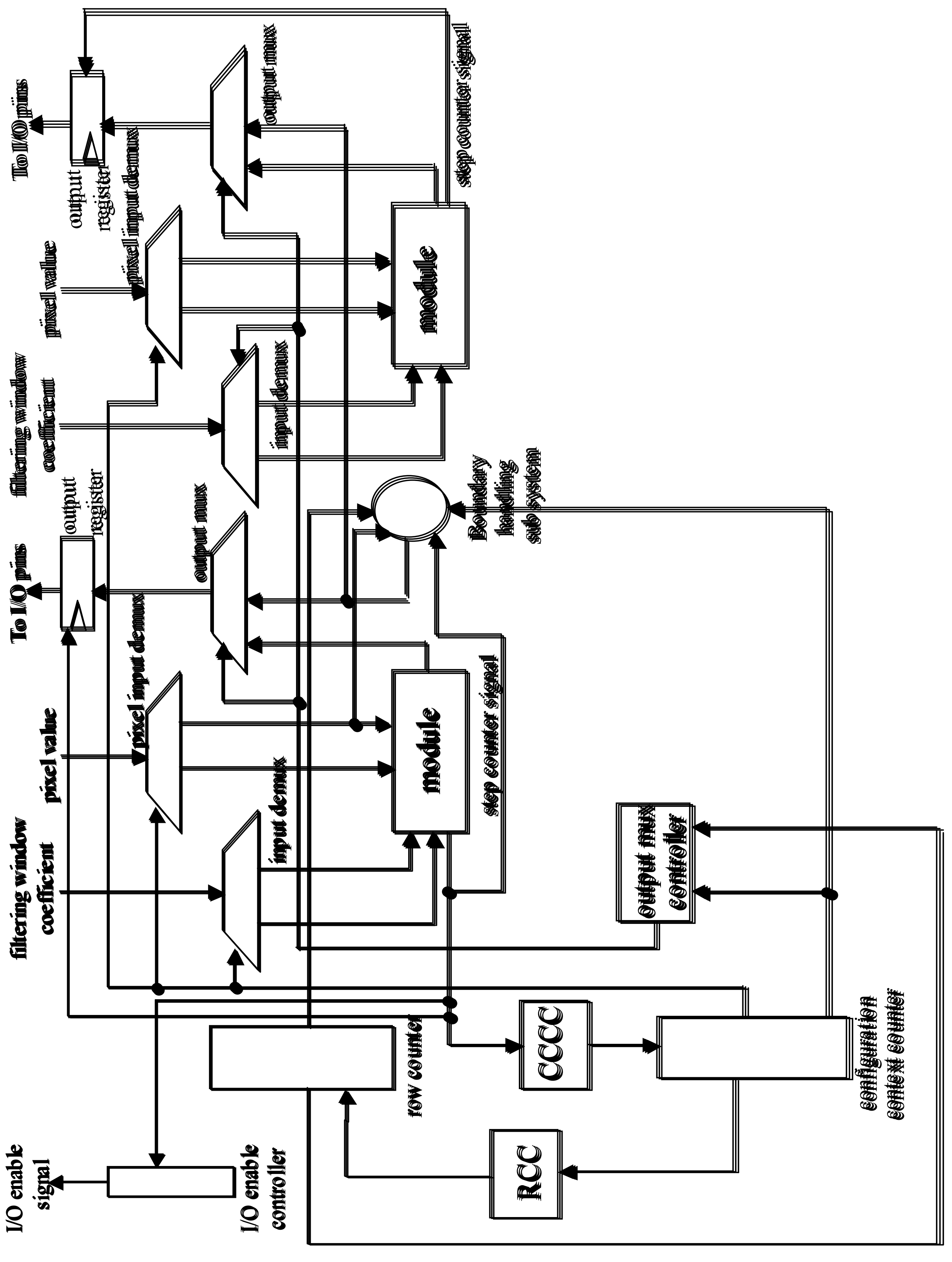

Figure 3.17 - I/O and control subsystem

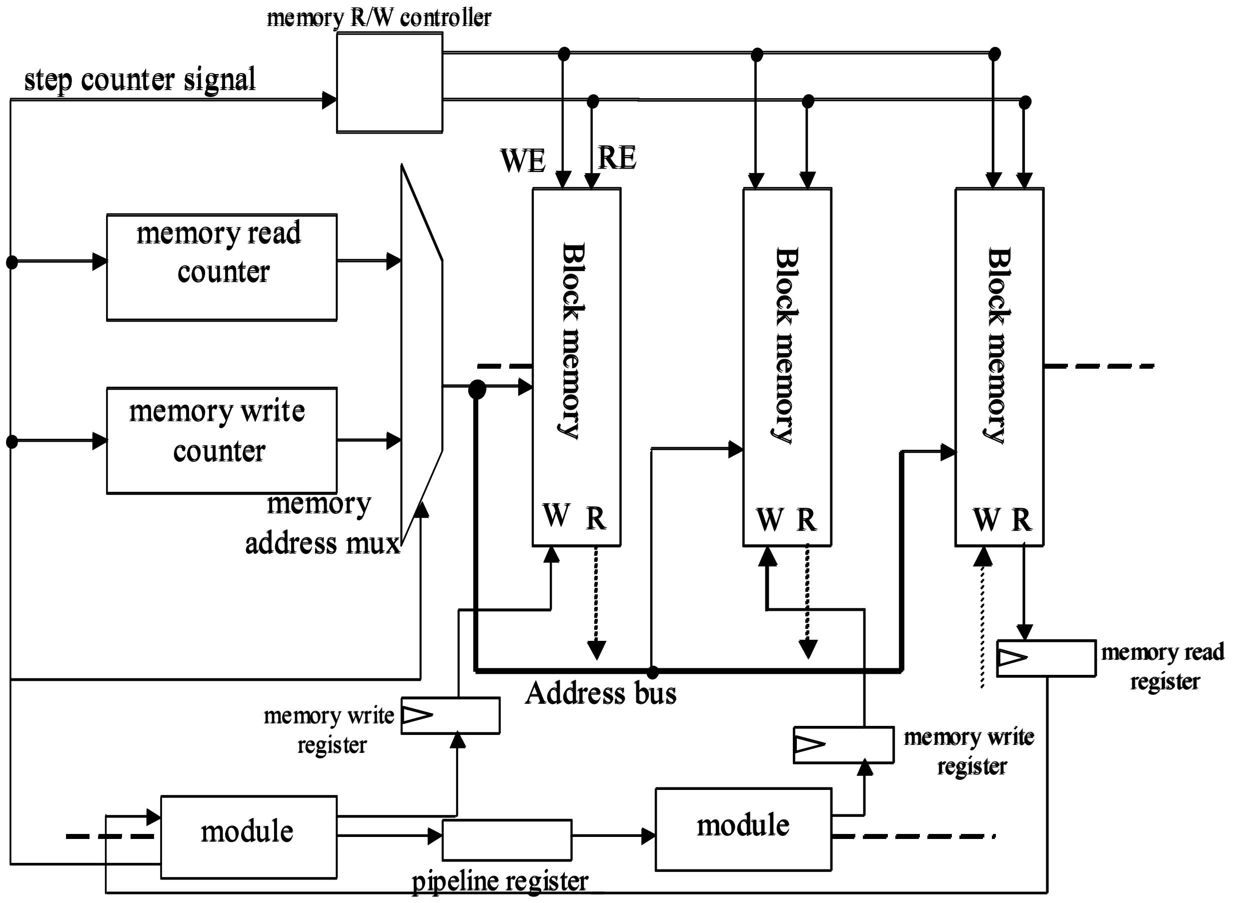

Figure 3.18 - Memory subsystem

We use several banks of 8-bit registers to store the pixel values of boundary pixels temporarily and pass them to the output pins at appropriate times. Controllers that receive their input signals from the row counter and the configuration context counter enable these registers.

.

Chapter 4: Results and Future Work

This chapter reports on the simulation and synthesis results of our implementation and provides some suggestions for future work at the end.

4.1 Results

We used VSIM to simulate the design and Leonardo Spectrum III to synthesize it on a Virtex™-E FPGA. The synthesis results for the design are as follows.

1. Number of I/O pins required = 384.
2. Number of CLB slices required = 492.
3. Clock frequency reported = 101.9 MHz, therefore clock length = 9.8 ns.

We now show the calculation for the time our implementation takes to process a 256×256 size gray scale image.

1. Time spent in one configuration context = clock length * 16 (recall that we have 16 steps, that is, 16 clock cycles per configuration context) = 9.8 * 16 = 156.8 ns.
2. Time spent in processing one row of the image = (time spent in one configuration context) * 16 = 156.8 * 16 = 2508.8 ns. (We have 16 configuration contexts per image row.)
3. Time spent in filtering the entire image = 256 * (time spent in processing one row) = 256*2508.8 = 642 μs.

We simulated the algorithm (spatially invariant linear filter algorithm) as a C program on two systems. Although our implementation handles adaptive filtering, it does not compute the adaptive window coefficients itself. Therefore, the comparison of running times between our implementation and a software-based spatially invariant linear filter algorithm gives us a correct measure of speedup. This is because the total numbers of computations performed are the same for both the adaptive RTR implementation and the spatially invariant filtering in software. Table 4.1 provides details of the software-based implementations reporting the running times and a comparison with our implementation, in terms of speedups that our implementation obtains.

Table 4.1 Running times of software-based implementations and speedups obtained with RTR implementation

System description	Running time	Speedup with RTR
866 MHz Pentium III system	20 ms	31
400 MHz Sun Ultra 5 system	53 ms	84

As reported in Table 4.1, the time taken by a Sun Ultra 5 system is inadequate for processing video images. A video image requires 25 images to be processed (filtered) per second, while the Sun system can process a maximum of only 19 images per second. Our implementation can process 1557, gray scale images of size 256×256 per second. As our implementation can easily scale for video images, it is suitable for processing video images as well.

4.2 Conclusion and Future Work

The speedups reported above prove that our implementation is superior in time to a software-based implementation of the algorithm. The speedup is due to the following reasons.

1. Our implementation is purely hardware-based.
2. Our implementation is parallel in nature as we process 16 pixels at the same time.
3. We use RTR in our implementation to further reduce the running time.

Some improvements are possible in this implementation that can reduce the running time.

1. We can reduce the number of configuration contexts and hence number of reconfigurations required by having 32 or more (a multiple of two) number of modules in the computation subsystem. This would lead to a greater degree of parallelism and reduce the number of configuration contexts required, leading to a decrease in the total number of idle cycles. This would require designing an advanced I/O subsystem as we are in this implementation curtailed by the number of I/O pins available on the device. We can reduce the total number of I/O pins per module from 24 to 16 or even to 8, with appropriate control.
2. Implementation of a 5×5 or larger size filtering window would also provide a greater speedup, as in the case of larger windows the time to reconfigure a configuration context would be less than the computation time. This would eliminate the seven idle clock cycles in the case of a 3×3 size filtering window.
3. Presently all computations of a configuration context are over in nine clock cycles. We wait for seven more clock cycles before reconfiguring a KCM so as to match the time required for reconfiguring the other KCM of the pair. This makes the control subsystem simple but leads to seven idle clock cycles per configuration context. Another approach can be to start reconfiguring a KCM after nine clock cycles instead of 16 (as currently done) and in this way use the seven idle clock cycles we have in every configuration context for reconfiguration. This approach would bring down the time spent during one computation-reconfiguration phase of a KCM to 25 clock cycles instead of the present 32 clock cycles. This would need a redesign of the control and I/O subsystem.
4. As a worthwhile addition to the application, generating filtering window coefficients on the FPGA itself would lead to a complete application running on the same FPGA.

References

[BP] K. Bondalapati and V. K. Prasanna, “Loop Pipelining and Optimization for Run Time Reconfiguration,” *Proc. Reconfigurable Architectures Workshop*, LNCS 1800, 2000, pp. 906-915.

[BR] S. Brown and J. Rose, “FPGA and CPLD Architectures: A Tutorial,” *IEEE Design & Test of Computers*, 1996, pp. 42-56.

[CCP] Y. Chung, S. Choi, and V. K. Prasanna, “Parallel Object Recognition on an FPGA-based Configurable Computing Platform,” *Int’l Workshop on Computer Architecture for Machine Perception*, Oct. 1997.

[CH] K. Compton and S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software,” *ACM Computing Surveys*, 2002, vol. 43, no, 2, pp.171-210.

[XIL05] K. Chapman, “Constant Coefficient Multipliers for XC4000E,” Xilinx Application Note, December 11, 1996 (Version 1.1).

[EM] H. Eng and K. Ma, “Noise Adaptive Soft-Switching Median Filter,” *IEEE Transactions on Image Processing*, vol. 10, no. 2, 2001, pp. 242-250.

[HCK] S. Hauck, “The Roles of FPGA’s in Reprogrammable Systems,” *Proceedings of the IEEE*, vol. 86, 1998, pp. 615-638.

[HW] B. Hutchings and M. Wirthlin, “Implementation Approaches for Reconfigurable Logic Applications,” *Proc.* 5th *International Workshop on Field-Programmable Logic and Applications*, 1995, pp. 419-428.

[JKS] Jain, Kasturi and Schmuck, Machine Vision, McGraw-Hill, New York, 1995, pp. 120-160.

[XIL04] D. Lim and M. Peattie, “Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations,” Xilinx Application Note: Virtex, Virtex-E, Virtex-II and Virtex-II Pro Families, May 17, 2002.

[IMG] G. Lacey and F. Shevlin, Computer Vision and Robotics Course, Computer Science Department, University of Dublin, Trinity College, *http://www.cs.tcd.ie/courses/computervision/ImageConvolutions.html*

[LS] J. Leonard and W. H. Mangione-Smith, “A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES,” *Proc. 7th Int’l. Workshop on Field-Programmable Logic and Applications*, 1997, pp. 151-160.

[LIM] J. Lim, “Image Enhancement” in *Digital Image Processing Techniques*, Academic Press, Orlando, FL, 1984, pp. 1-51.

[LM] Z. Luo and M. Martonosi, “Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques,” *IEEE Transactions on Computers*, vol. 49, no. 3, March 2000, pp. 208-218.

[RGV] J. Rose, A. Gamal, and A. Vincentelli, “Architecture of Field-Programmable Gate Arrays,” *Proceedings of the IEEE*, vol. 81, 1993, pp. 1013-1029.

[SLBC] N. Shirazi, W. Luk, D. Benyamin, and P. Y. K. Cheung, “Quantitative Analysis of Run-Time Reconfigurable Database Search,” *Proc. 9th Int’l. Workshop on Field-Programmable Logic and Applications*, 1999, pp. 253-263.

[TBW] C. Tanougast, Y. Berviller, and S. Weber, “Optimization of Motion Estimator for Run-Time-Reconfiguration Implementation,” *Proc. Reconfigurable Architectures Workshop*, LNCS 1800, 2000, pp. 959-965, 2000.

[TKP] A. M. Tekalp, *Digital Video Processing*, Prentice Hall PTR, Upper Saddle River, NJ, 1995, pp. 262-282.

[VH] J. Villasenor and B. Hutchings, “The Flexibility of Configurable Computing,” *IEEE Signal Processing Magazine*, vol. 15, no. 5, 1998, pp. 67-84.

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

[WE] M. Wojko and H. Elgindy, “Configuration Sequencing with Self Configurable Binary Multipliers,” *Proc. 6th Reconfigurable Architecture Workshop* (Parallel and Distributed Processing; Lect. Notes Comp. Science. #1586), pp. 643-651.

[XIL01] Xilinx, Inc., “Virtex™-E 1.8 V Field Programmable Gate Arrays,” Xilinx Production Product Specification, September 10, 2002.

[XIL02] Xilinx, Inc.,“Virtex Series Configuration Architecture User Guide,” Xilinx Application Note: Virtex Series, September 27, 2002.

[XIL03] Xilinx, Inc., “Using the Virtex Block SelectRAM Features,” Xilinx Application Note: Virtex Series, December 18, 2000.

[XIL06] Xilinx, Inc., “Constant (k) Coefficient Multiplier Generator for Virtex,” Xilinx Application Note, March 21, 1999.

Vita

Nitin Srivastava was born in the city of Varanasi in India. At the age of five his parents moved to the city of Lucknow, the capitol city of the state of Uttar Pradesh. He graduated from high school in the year 1992. In the fall of 1993, he joined the Regional engineering College at Warangal and obtained his Bachelor of Technology degree in Electrical Engineering in June 1997 in first divison.

From September 1997 to March 1999, he worked as a software engineer in MBT Ltd. at Pune, India. Later he joined British Telecom, Plc. as a system administrator in April 1999 and lived and worked in the city of London, England until December 2000.

He joined Louisiana State University in the spring of year 2001 and is set to obtain his Master of Science in Electrical Engineering degree in May 2003.