Flexible Systolic Architecture for Image Processing at Video Rate

by

M. C. Griselda Saldaña González

A Dissertation submitted to the program in Computer Science,
Computer Science Department
in partial fulfillment of the requirements for the degree of

DOCTOR IN COMPUTER SCIENCES

at the

National Institute of Astrophysics,
Optics and Electronics
2007
Tonantzintla, Puebla

Advisor

Dr. Miguel Octavio Arias Estrada
Principal Research Scientist
Computer Science Department
INAOE

Flexible Systolic Architecture for Image Processing at Video Rate

by
Griselda Saldaña
Thesis
Submitted to the Department of Computer Science,
on 9th July 2007, in partial fulfillment of the
requirements for the degree of
DOCTOR IN COMPUTER SCIENCE

Abstract

Computer performance has improved tremendously since the development of the first all-purpose, all electronic digital computer in 1946. However, engineers, scientists and researchers keep making more efforts to further improve the computer systems performance to meet the demanding requirements for many applications such as Computer Vision and Image Processing which requires a high computational power to solve data-intensive applications in real-time.

There are basically three ways to improve the computer performance of algorithms in terms of computational speed. One way is *increasing the clock speed*; this parameter is determined by the worst-case delay in the datapath. The datapath elements can be rearranged such that the worst-case delay is reduced. Furthermore, it is possible to reduce the number of datapath actions taken in a single clock cycle. However, such attempts at reducing the clock cycle time have an impact on the number of Clocks per Instructions (CPIs) needed to execute the different instructions. Another way is to *reduce the CPI* by increasing the hardware concurrency. The final option consists in *reducing the number of instructions*; for this purpose it is possible to replace simple instructions by more complex ones so that the overall program executes fewer instructions. Once again any attempt to introduce new complex instructions has to be carefully balanced against the CPI, the clock cycle time, and the dynamic instruction frequencies.

Special-purpose parallel systems and, in particular the ones referred to as systolic arrays are very attractive approaches for handling many computationally-intensive applications. These systems consist of an array of identical Processing Elements (PE) executing the same operations on a set of data. These arrays capitalize on regular, modular, rhythmic, synchronous, concurrent processes that require intensive, repetitive computations.

The main obstacle to the widespread use of application-specific arrays of processors is development time, cost and their capacity to support a single algorithm at the same time. Recently, the use of reconfigurable hardware devices in the form of Field Programmable Gate Arrays (FPGAs) has been proposed as a means to implement parallel high performance solutions at an affordable price. These circuits provide a homogeneous surface of general-purpose logic elements which can be configured as often as desired to implement any combinational or sequential circuit.

Parallel processing architectures based on FPGAs provide an alternative to faster clock performance. This characteristic turns this approach into an attractive tool for high performance architecture implementation.

Low-level image processing operators play a fundamental role in modern image processing and computer vision algorithms. These operators exhibit natural parallelism that can be easily exploited using array of processors implemented with FPGAs.

Traditionally systolic array implementations are special purpose since they fit to one special algorithm; however in order to provide a higher degree of flexibility and generalization certain level of programmability support is essential. There have been previous efforts to develop general purpose systolic arrays; however these implementations require large local memories, and high-bandwidth for data communication between processors and global memory.

Within this context, this dissertation addresses the design and development of an FPGA-based image processing hardware architecture using a simple, resource-limited systolic array. The architecture is aimed to support operations involved in common low-level image processing algorithms which include 2D convolution, image filtering, matrix-matrix multiplication, morphological operations and pyramid processing. Furthermore, the architecture has been designed to pursue the implementation of higher complexity algorithms such as motion estimation, which has also been implemented in order to verify the generalization of the proposed schema. The architecture can achieve a processing rate that allows performance in real time, consuming a small amount of area. This feature allows replicating the architecture modules inside the same FPGA several times, consuming a small quantity of power.

The parallelism of this architecture has been explored for different key parameters. Different blocks in the architecture have been developed to generate a variety of operations, with different tradeoff in size and performance based on user-defined parameters. These parameters include the bit-width, array size, window size, image size and the application to be performed. This set of parameters determines the complexity of the operations in hardware, performance, power consumption, reliability and area occupancy which can guide for tradeoff during implementations for a particular application.

Memory optimization has been done at architectural-level in order to meet the best area-speed-power tradeoff. This was achieved by reducing the amount of memory accesses through memory splitting into buffers. In the context of memory bandwidth, an efficient balance between on-chip and off-chip memory has to be obtained to meet the best power-bandwidth tradeoff.

A complete image processing system usually requires a sequence of different tasks to be performed on an image. The intermediate result of one task is just the input of the

next one. This process requires that data processed in the first stage are reused in subsequent stages. To facilitate the movement of the information among different phases of processing, in the proposed architecture Router elements are employed.

Using Routers it is possible to direct data between processing blocks that perform different algorithms inside the same architecture to chain results from these different processing stages. Furthermore Routers improve the system scalability constituting a means to increase the number of processing blocks inside the system

From high level, the architecture resembles a pipeline schema where a group of buffers stores data to be sent to the next processing block via Routers elements. This schema highly improves the architecture flexibility, since data movements among processing stages can be defined by the user.

The main contributions of the thesis are the following:

- Proposal of a new, high performance, flexible hardware architecture specialized in low-level processing under real time, which implies a processing rate of at least 30 frames per second, required in most of video standards.
- Generalization of the architecture as a hardware platform capable for test and implementation of higher complexity algorithms.
- Implementation of an enhanced systolic array that overcomes its inherent constraints such as extensibility (in the sense that it is impossible to produce an array to match all the possible sizes of different problems), speed limitation and high latency for large arrays.
- Implementation of a mechanism for easy scalability that allows enhancing the system by adding modules without redesigning the current basic structure.
- Implementation of a mechanism to chain processes in order to solve higher complexity algorithms.

- Analysis of area-speed-power tradeoff between the architecture main parameters.
- Analysis of constraints in area-performance behavior.

Arquitectura Sistólica Flexible para el Procesamiento de Imágenes a Velocidad de Video

por
Griselda Saldaña
Tesis
Sometida al Departamento de Ciencias Computacionales,
el 9 de julio de 2007, como requisito parcial
para la obtención del grado de
DOCTOR EN CIENCIAS COMPUTACIONALES

Resumen

El desempeño computacional se ha mejorado tremendamente desde el desarrollo de la primera computadora totalmente electrónica de propósito general en 1946. Sin embargo, los ingenieros, científicos e investigadores siguen realizando esfuerzos que permitan obtener mejoras en el desempeño de los sistemas computacionales para satisfacer las exigencias de muchas aplicaciones tales como la Visión por Computadora, que requiere un alto poder computacional en la resolución de aplicaciones intensivas en datos bajo condiciones de tiempo real.

Básicamente hay tres formas de mejorar el desempeño de los algoritmos en términos de velocidad computacional. Una forma consiste en aumentar *la velocidad del reloj*, este parámetro se determina por el peor retardo en el datapath. Los elementos datapath se pueden reorganizar de manera tal que el retardo se reduzca. Además, es posible reducir el número de acciones que realiza el datapath en un solo ciclo de reloj. Sin embargo, tales tentativas por reducir el tiempo de ciclo de reloj tienen un impacto sobre el número de Ciclos por Instrucciones (CPIs) necesarias para ejecutar las diferentes instrucciones. Otra forma consiste en *reducir los CPIs* aumentando la concurrencia de hardware. La opción final consiste en *reducir el número de instrucciones*, para este propósito es posible reemplazar instrucciones simples por instrucciones más complejas de tal manera que el programa total ejecuta menos instrucciones. De nueva cuenta, cualquier tentativa por introducir nuevas

instrucciones complejas tiene que ser cuidadosamente balanceada contra los CPIs, el tiempo del ciclo de reloj y las frecuencias de instrucciones dinámicas.

Los sistemas paralelos de propósito específico y en particular los conocidos como arreglos sistólicos resultan ser enfoques muy atractivos para el manejo de muchas aplicaciones computacionalmente intensivas. Estos sistemas consisten de un arreglo de Elementos de Procesamiento (PE) idénticos que ejecutan la misma operación sobre un conjunto de datos. Estos arreglos toman ventaja de los procesos concurrentes, regulares, modulares, rítmicos, sincrónos, que requieren cálculos repetitivos e intensivos.

Los principales obstáculos para extender la utilización de los arreglos de procesadores de aplicación específica son el tiempo de desarrollo, el costo y la capacidad que tienen de dar soporte a un solo algoritmo a la vez. Recientemente, el empleo de dispositivos de hardware reconfigurables en forma Arreglos de Compuertas Programables en Campo (FPGA) se ha propuesto como un medio para implementar soluciones paralelas de alto rendimiento a un bajo costo. Estos circuitos proporcionan una superficie homogénea de elementos lógicos de propósito general que se pueden configurar tan a menudo como sea necesario para implementar cualquier circuito combinacional o secuencial.

Las arquitecturas de procesamiento en paralelo basadas en FPGAs proporcionan una alternativa para un desempeño más rápido del reloj. Esta característica convierte a este enfoque en un instrumento atractivo para la puesta en práctica de arquitectura de alto rendimiento.

Los operadores de procesamiento de imágenes de bajo nivel juegan un papel fundamental en los algoritmos de procesamiento de imágenes y de visión por computadora modernos. Estos operadores presentan un paralelismo natural que pude

ser aprovechado fácilmente empleando arreglos de procesadores implementados con FPGAs.

Tradicionalmente, los arreglos de procesadores sistólicos son de propósito específico dado que sólo resuelven un algoritmo determinado, sin embargo, para proporcionarles un mayor grado de flexibilidad y generalidad es necesario proporcionar cierto nivel de programabilidad. Previamente se han realizado esfuerzos para desarrollar arreglos sistólicos de propósito general, sin embargo, estas implementaciones requieren memorias locales grandes y un gran ancho de banda para la comunicación de datos entre los procesadores y la memoria global.

Dentro de este contexto, este trabajo de investigación aborda el diseño y desarrollo de una arquitectura hardware versátil, basada en tecnología FPGA, para el procesamiento de imágenes, empleando un arreglo sistólico simple y recursos limitados. La arquitectura tiene como objetivo dar soporte a las operaciones involucradas en algoritmos de procesamiento de bajo nivel de imágenes comunes, entre los que se encuentran convolución 2D, filtrado, multiplicación de matrices, operaciones morfológicas y procesamiento piramidal. Además, la arquitectura se ha diseñado buscando la implementación de algoritmos de mayor complejidad tales como la estimación de movimiento, que también ha sido implementada para verificar la generalización del esquema propuesto. La arquitectura puede alcanzar una razón de procesamiento que permite desempeño en tiempo real, consumiendo una pequeña cantidad de área. Esta característica permite replicar los módulos de la arquitectura varias veces dentro del mismo FPGA, consumiendo una cantidad de potencia reducida.

En esta arquitectura se ha explorado el paralelismo para los diferentes parámetros principales. Diferentes bloques en la arquitectura se han desarrollado para generar una variedad de operaciones, con diferentes compromisos en tamaño y desempeño con base en parámetros definidos por el usuario. Estos parámetros incluyen el número

de bits, el tamaño del arreglo, el tamaño de la ventana, el tamaño de la imagen y la aplicación a ser realizada. Este conjunto de parámetros determina la complejidad de las operaciones en el hardware, el desempeño, el consumo de potencia, la confiabilidad y la ocupación de área que proporcionan una guía para determinar los compromisos durante la implementación de una aplicación en particular.

Se ha realizado optimización de memoria a nivel arquitectural para encontrar el mejor compromiso entre área-velocidad-potencia. Esto se pudo alcanzar reduciendo la cantidad de accesos a memoria dividiéndola en buffers. En el contexto de ancho de banda de memoria, se debe obtener un balance eficiente entre la memoria interna y la memoria externa para encontrar el mejor compromiso entre potencia-ancho de banda.

Un sistema de procesamiento de imágenes completo por lo general requiere realizar una secuencia de tareas diferentes sobre una imagen. El resultado intermedio de una tarea es la entrada de la siguiente. Este proceso requiere que los datos procesados en la primera etapa sean reutilizados en etapas subsecuentes. Para facilitar el movimiento de la información entre diferentes fases de procesamiento, en la arquitectura presente se utilizan elementos Ruteadores.

Utilizando Ruteadores es posible dirigir los datos entre bloques de procesamiento que realizan algoritmos diferentes dentro de la misma arquitectura para encadenar resultados de etapas de procesamiento diferentes. Además los Ruteadores mejoran la escalabilidad del sistema ya que constituyen un medio para incrementar el número de bloques de procesamiento dentro del sistema.

Vista en alto nivel, la arquitectura asemeja un esquema de pipeline donde un grupo de buffers almacenan datos para ser enviados al siguiente bloque de procesamiento vía elementos de Ruteadores. Este esquema mejora la flexibilidad de la arquitectura ya que el movimiento de los datos entre etapas de procesamiento puede ser definido por el usuario.

Entre las principales contribuciones de este trabajo se puede mencionar:

- Propuesta de una nueva arquitectura flexible de alto desempeño, especializada para el procesamiento de bajo nivel de imágenes en tiempo real, lo que implica una razón de procesamiento de al menos 30 cuadros por segundo, requerido en la mayoría de los estándares de video.
- Generalización de la arquitectura para conformar una plataforma hardware que permita la implementación y prueba de algoritmos de mayor complejidad.
- Implementación de un arreglo sistólico mejorado que supera las restricciones inherentes a este tipo de elementos, tales como la extensibilidad (en el sentido de que es imposible producir un arreglo para todos los tamaños posibles de los diferentes problemas), limitación de velocidad y una alta latencia en arreglos grandes.
- Implementación de un mecanismo para un fácil escalamiento del sistema que permite realizar mejoras agregando módulos sin tener que rediseñar la estructura básica actual.
- Implementación de un mecanismo de encadenamiento de procesos para resolver algoritmos de mayor complejidad.
- Análisis del compromiso entre área-velocidad-potencia entre los principales parámetros de la arquitectura
- Análisis de las restricciones en el comportamiento área-desempeño

Acknowledgments

To the CONACYT for giving me the financial support to belong to an institution of excellence as the INAOE.

To the INAOE for providing the infrastructure and the human resources that allowed me to finish my PhD studies.

Furthermore, I want to take the opportunity to thank to my advisor Dr. Miguel Octavio Arias Estrada for his constant direction, supervision and support during the development of this thesis. His wide knowledge and his clear view have been of great value for the development of this research work. His ideas and concepts have had a remarkable influence on my entire career.

I want to thank also to the other members of my PhD committee who monitored my work and took effort in reading and providing me with valuable comments on this thesis: Dra. Claudia Feregrino Uribe, Dr. René Armando Cumplido Parra, Dr. Leopoldo Altamirano Robles, Dr. Carlos Alberto Reyes García, and Dr. Oswaldo Cadenas.

Furthermore I want to thank to all the members of the Computer Science Department of the INAOE who gave the opportunity to work with them in a good working atmosphere.

Dedicatorias

A mis padres

Emilio Saldaña Márquez
Blanca González Vázquez

Y a mis hermanos

Sara, Blanca y Emilio

Por apoyarme con su amor, paciencia y comprensión; por animarme a alcanzar todas mis metas profesionales. Ustedes son mi motivación y mi fuerza.

A mi cuñado Héctor de la Rosa por su apoyo y compañerismo.

A Gabriel Coronado por su paciencia y constancia durante los momentos más extenuantes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Real-time image processing systems are finding many new applications in areas such as real time video rate processing, medical systems, multimedia and mobile systems. In building these systems, designers have essentially three options: developing the vision algorithms in software and running them on a *standard processor*, designing custom hardware specially tailored for the application like *Application Specific Integrated Circuits* (ASICs), or the use of *programmable hardware*.

The processing of images by computer is time costly due to the number of operations required and the volume of data to be processed [1]. The task is even more complex when the processing has to be carried out in real time. In order to meet the desired requirements of achieving the highest performance at lowest cost under real-time, special-purpose hardware architectures are often used. For performance improvement, reconfigurable computing systems have demonstrated to be a valuable alternative to traditional implementations. Reconfigurable solutions can often be orders of magnitude faster and/or less expensive than conventional alternatives like ASICs [2, 3]. Additionally reconfigurable systems have several advantages such as reduced development time; the capability to alter the hardware circuit after the system has been deployed in the field, and the possibility to implement modular and scalable applications.

General Purpose Processors (GPPs) are flexible enough to implement a variety of applications using the same device; however they can not always offer the computational power required for providing fast implementation of computationally intensive operations. In custom designed circuits functionality is hardwired once and cannot be changed again. ASIC implementations are usually efficient and can perform operations faster than other approaches; furthermore ASICs take advantage of existing parallelism in many image processing algorithms. But they are not implemented in practice because the required resources are not usually available at an affordable price and the design process takes a long time. These facts have usually limited researchers to the former option, which is to develop algorithms that can be executed as fast as possible on standard processors.

In the last few years, the third solution for real-time image processing system designers has become viable due to the rapid growth in capacity and speed of programmable hardware, which has demonstrated its efficiency to execute complex algorithms satisfying the simultaneous demand for application performance and flexibility [4, 5]. Programmable hardware has also been used successfully in some non signal processing applications [6]. FPGA technology allows designers to configure the chip according to the specifications of the algorithm cheap and quickly because it eliminates the most expensive and time consuming part of ASIC fabrication. It also reduces the debugging time because one can typically re-compile the design in a few hours and reconfigure the FPGAs in less than a second. FPGA technology presents four main benefits:

1. **Performance**. FPGAs enable custom computing systems to be highly specialized to specific data, as well as specific applications. One typical optimization is to implement highly parallel architectures that can exploit significant data-level parallelism. By capitalizing on these opportunities, a highly programmable system can be constructed that attains near ASIC performance.

2. **Cost Effectiveness**. Configurable computing can be used to reduce system costs through two approaches: hardware reuse and low Non-Recurring Engineering. A number of research efforts have demonstrated time-shared methods for simulating a large circuit on a smaller FPGA. Furthermore, as the feature size of semiconductor processes shrink FPGAs technology become much more cost effective.
3. **Custom I/O**. FPGAs provide an extremely rich and flexible set of programmable I/O signals. These components give the system designer an opportunity to reuse existing hardware, or commit earlier to a new hardware design. The benefits of custom I/O extend beyond the prototype stage; for example system functionality may change after deployment, and the ability to rewire system I/O through reprogramming the FPGA can be an invaluable advantage.
4. **Fault-Tolerance**. Some recent efforts have investigated the benefits of using FPGAs to provide fine-grained fault recovery. This approach has the benefit of increasing system reliability for much lower cost than the traditional approach of circuit and subsystem redundancy.

Low-level image processing operators play a fundamental role in modern image processing and computer vision algorithms. These operators exhibit natural parallelism that can be easily exploited using array of processors implemented with reconfigurable hardware.

FPGA implementations of this kind of applications have the potential to be parallelized using a mixture of spatial and temporal parallelism. Pragmatically, the degree of parallelization is subject to the processing mode and hardware constraints imposed by the system. Based on previous work [7-9] three main constraints have been identified for implementation: timing (limited processing time), bandwidth (limited access to data), and resource (limited system resources).

- **Timing constraints**. The data rate requirements of the application impose a timing constraint which in turn drives the other constraints, when real-time is demanded this restriction become crucial. If there is no requirement on processing time then the constraint on bandwidth is eliminated because random access to memory is possible and desired values in memory can be obtained over a number of clock cycles with buffering between cycles.
- **Bandwidth constraints**. Some operations require that the image be partly or wholly buffered because the order that the pixels are required for processing does not directly correspond to the order in which they are input. Frame buffering requires large amounts of memory. Typically FPGAs use off-chip memory for frame buffering but this may only allow a single access to the frame buffer per clock cycle, which can be a problem for the many operations that require simultaneous access to more than one pixel from the input image.
- **Resource constraints**. This issue arises due to the finite number of available resources in the system such as local and off-chip RAM, or other function blocks implemented on the FPGA. Programming without consideration of the hardware that will be generated has a direct effect on the speed of the implementation.

These constraints are inextricably linked and manifest themselves in different ways depending on the processing mode.

Traditionally these issues have been faced developing application specific architectures optimized to be used in the systems for which they were designed. In order to improve performance some well known pipeline structures have been used. They help to reduce memory overhead when predictable scanning schemas are used; however, it turns out that they cannot cope with unpredictable image scanning which has proved to be very efficient in the implementation of certain operators. Furthermore pipelining results in an increase in logic block usage, caused by the need to construct pipeline stages and registers. While the impact of pipeline registers on logic block usage will be minimal, care must still be taken to make efficient use of the available logic blocks.

This thesis aims to make progress on the question of whether reconfigurable computing will supply a viable option to fulfill the requirements of real-time image processing systems of high performance, high flexibility and low power consumption. For this purpose it is essential to explore new design and implementation strategies at the architectural level. Also it is relevant to deal with issues arising from the integration of reconfigurable hardware with a processor including:

- To make an efficient use of area avoiding a poor utilization of the silicon resources available.
- To reduce latencies due to increased communication.
- To avoid a reduced bandwidth looking for scalability.
- Smart use of memory.
- To implement a cost-effective reconfigurable processor for intensive computer vision applications applying reconfigurable techniques.
- To achieve a good tradeoff in computer vision system design under the Reconfigurable Computing approach.

Many complex image processing algorithms use low-level results of window operators as primitives to pursue higher level goals [10, 11]; thus another problem to be addressed consists of finding a way to extend the architecture capability and flexibility to support these higher level applications. In this case, several issues must be taken into account:

- To use a reduced amount of on-chip memory for buffering data.
- Low cost of reconfiguration.
- Smart parameters transfer.
- High potential for scalability.

1.2 Objectives and Contribution

The main objective of this thesis is to demonstrate that making use of parallelism of data present in low-level image operators, and reconfiguration techniques it is possible to implement a versatile reconfigurable systolic architecture for low-level image processing achieving high performance and low power consumption in real time.

The research aims to build a system for such commitments providing the processing of 30 frames per second on a 640×480 sized grey level images:

- To implement a high performance flexible architecture specialized in low-level processing under real time.
- To propose a parallel architecture based on reconfigurable modules to execute a sequence of algorithms according to a predefined pattern.
- To implement a customizable systolic array allowing the use of different window size in image operations.
- To analyze area-speed-power tradeoff between the architecture's main parameters.
- To optimize architecture performance according to metrics like:
    - Processing Time
    - Area Used
- To propose guidelines to apply Hardware/Software co-design in the proposed architecture.
- To define a testbench to quantify the improvement obtained with the proposed reconfigurable architecture compared to previous approaches.

The research approach and the key contributions of the thesis are as follows:

- Design and development of a new flexible high performance FPGA-based hardware architecture for window-based image processing, that can be generalized to support other algorithms of higher complexity.
- Implementation of a systolic array with extended capacities to reduce its inherent constraints such as extensibility (in the sense that it is impossible to produce an array to match all the possible sizes of different problems).
- Determining metrics to reduce the reconfiguration cost associated to reconfiguration techniques.
- Implementation of a mechanism to chain processes in order to solve algorithms of higher complexity.
- Analysis of mechanisms to build a hardware library of reusable image processing modules.
- Analysis of potential application for the architecture and its implications for implementation.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 gives a background to the work undertaken in the thesis. It explains the concept of reconfigurability upon which this research is based, along with a review of other works that led to this research. Some mechanisms of reconfiguration are described in detail to identify its major characteristics. Previously proposed hardware implementations of image processing tasks are briefly analyzed and discussed in order to identify some problems, limitations and drawbacks. This chapter details the different FPGA approaches for image processing hardware implementations and concludes presenting some unsolved problems to be addressed using reconfigurable computing.

Chapter 3 reviews different characteristics of parallel architectures for image processing discussing in detail the concepts of systolic processing, pipeline and Single Instruction Multiple Data (SIMD) in the context of image processing as well as their application to a special kind of operators denominated as window-based operators and concludes with some considerations for an FPGA implementation of such systems.

Chapter 4 introduces the systolic architecture for window-based image processing providing its functional requirements as well as a top-down description of its main modules and their operation. The data movement and the memory schema used are presented to highlight the mechanism to reduce number of access to memory.

Chapter 5 provides the FPGA implementation details of the architecture and a description of the hardware used to test and to implement the system. Furthermore an analysis of the considerations for the implementation of fixed-point operations is performed in order to achieve a small amount of error. Simulation and synthesis results, timing and performance of the proposed architecture are presented using standard metrics for evaluation; a special section of this chapter is devoted to the performance analysis and discussion of the architecture and the comparison with other technologies. A brief discussion is carried out at the end of the chapter highlighting the main features of the architecture; modularity, communication, memory and data flow.

Chapter 6 presents some processing applications mapped to the architecture: convolution, filtering, matrix multiplication, pyramid decomposition, morphological operators, and template matching to validate the correct functionality of the proposed architecture. One section of this chapter is devoted to determine the silicon area occupied by the architecture in order to use this parameter as a standard metric for evaluation. The synthesis results obtained for each operator are presented to make a comparison with previous systems implemented, highlighting the main improvements achieved with the present implementation.

Chapter 7 describes some concepts of motion estimation algorithms from a video coding perspective, focusing on block based techniques; furthermore a short review of previous work in literature is presented. In this chapter some characteristics of motion estimation algorithm that allow the representation as a window operator are clarified to determine the modifications required to the implementation of this algorithm. The synthesis results for this version of the architecture are presented as well as a discussion of the performance achieved.

Finally chapter 8 presents the thesis conclusions emphasizing the advantages provided by the implementation of the present architecture in comparison with previous works, the accomplishments and contributions are detailed. In addition some potential applications and extensions for the architecture are discussed in order to outline the future work. In this section the possibility of using dynamic reconfiguration is remarked due to this is a strong research direction to follow.

Chapter 2

Background and Previous Work

There are two traditional approaches to implement digital logic systems: mapping an algorithm to a General Purpose Processor and designing custom hardware that implements an algorithm. GPPs can implement a wide variety of tasks, but do not fully utilize the potential power of the silicon with which they are implemented. The other approach is to design custom silicon for a particular task. The custom silicon is commonly known as an ASIC.

ASICs are designed specifically to perform a given computation, and thus they are very fast and efficient when executing the exact computation for which they were designed. However, the circuit cannot be altered after fabrication. This forces a redesign and refabrication of the chip if any part of its circuitry requires modification. This is an expensive process, especially when one considers the difficulties in replacing ASICs in a large number of deployed systems.

ASICs can be used to carry out fixed applications that have to be executed with the minimum amount of area, delay and energy costs. As the size of the application grows, it becomes practically impossible to implement it in silicon. This is where the GPP steps in. Processors execute a set of instructions to perform a computation. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance may diminish, if not in clock speed then in work rate, and is far below than an ASIC. The processor must read each instruction from memory, decode its meaning, and only then execute it. This results in a high execution overhead for each individual operation. Additionally, the set of instructions that may be used by a program is determined at the fabrication time of the processor. Any other operations that are to be implemented must be performed out of existing instructions.

The reconfigurable computing domain has emerged as a new computing paradigm to fill the gap between ASICs and GPPs [12], achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. The goal of reconfigurable systems is to achieve implementation efficiency approaching that of specialized logic while providing the silicon reusability of general purpose processors [13].

FPGAs are the building blocks for reconfigurable computing. The ability to exploit the parallelism often found in the algorithms, as well as the ability to support different modes of operation on a single hardware substrate, gives to these devices a particular advantage over fixed architecture devices such as serial Central Processing Units (CPUs) and Digital Signal Processors (DSPs). Furthermore, development times are substantially shorter than dedicated hardware in the form of ASICs, and small changes to a design can be prototyped in a matter of hours. Adding a reconfigurable portion to a system, enables it to support a broader range of applications and lets developers adapt the hardware to changing needs and evolving standards. Thus a reconfigurable platform offers the advantages of both embedded software and custom hardware.

For optimum implementation of algorithms used in high-performance computing applications, reconfigurable hardware is a potential option to be applied. FPGAs are a more effective solution because the architecture can be defined at runtime. Both connectivity and processing capabilities can be tailored to suit the needs of the algorithm being implemented.

In particular, the largest part of the die area of an FPGA is consumed by the configurable routing and this is important because in general, the biggest limiting factor of any implementation of a parallel algorithm is data passing [14]. For this reason FPGAs will always offer significant opportunities of improvement over any GPP with fixed communication buses. The majority of the silicon used by a GPP is not actually operating on each clock cycle because there is generally only one instruction being executed at a time and each of those instructions rarely use a significant fraction of the capability of the processor. FPGAs offer a big advantage here not only because all of the hardware can be executing on each clock cycle, but also because the mix of computational elements can be tuned to the precise requirement of the application.

For these reasons, among others, reconfigurable computing offers significant advantages over fixed computing and these advantages are overwhelming in the area of high performance computing such as Video Processing [14].

2.1 FPGAs

The most common reconfigurable devices today are Field Programmable Gate Arrays. An FPGA is a programmable logic device that supports implementation of logic circuits, its real advantage is that the chip can be erased and re-programmed any number of times making the process of debugging both quick and cheap, requiring just one initial purchase cost.

Traditionally, FPGAs were used primarily for prototyping. A designer could quickly prototype his designs in hardware to check that the design meets the specification and provides a correct solution. Through this use, it is unnecessary, until the final stages of prototyping, to ever fabricate the hardware design being developed. This drastically reduces both the cost and the time of hardware development [15]. In recent years, both the performance and capacities of FPGAs have grown such that they can now be used to implement many end products. While the process of moving the FPGA into commercial products has been used in embedded systems for a while, it is only now beginning to be used in non-embedded computing [16].

Typical architecture of a Xilinx FPGA comprises a regular array of Configurable Logic Blocks (CLBs) with routing resources for interconnection and it is surrounded by programmable Input/Output Blocks (IOBs). CLBs provide the functional elements for constructing logic while IOBs provide the interface between the pins of the package and the CLBs. FPGAs are widely used as a prototype before fabricating a VLSI design, or can be used directly in a product. Figure 2.1 shows the basic structure of Xilinx FPGAs [17].

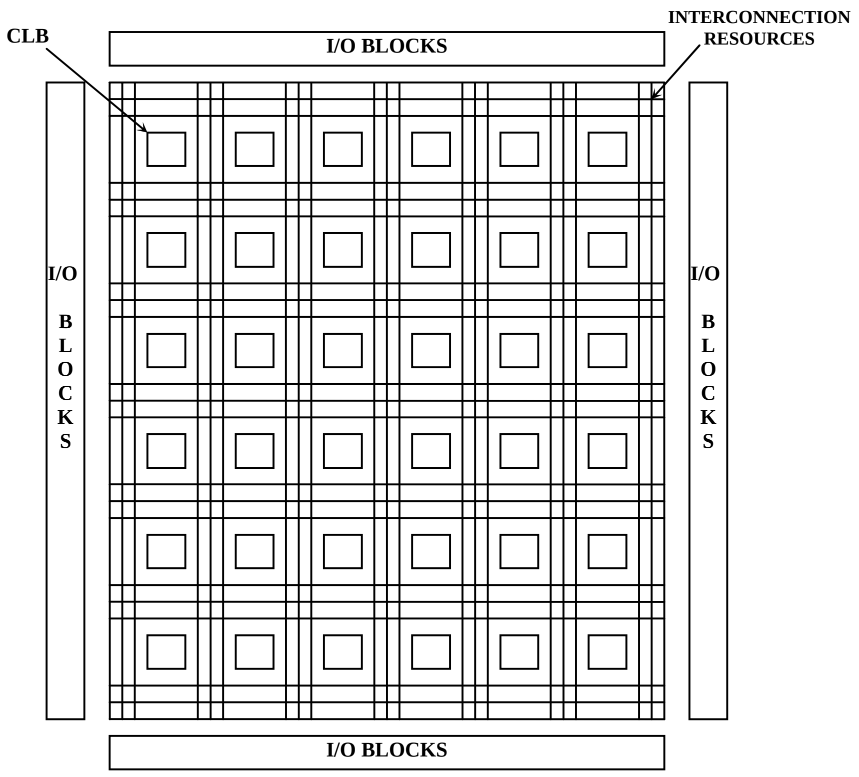


Figure 2.1 Basic structure of Xilinx FPGAs.

The structure of a CLB can be as simple as a transistor or as complex as a microprocessor [18]. The CLBs can be arranged in a row or, more commonly, in a matrix form. The number of CLBs available also varies from vendor to vendor. The interconnection network serves as the underlying fabric to provide flexible interconnection between CLBs for logic synthesis. Of the three building blocks in an FPGA, the interconnection network typically occupies maximum space [19].

A CLB is constructed from the following components:

1. **Look up tables (LUT):** A CLB contains a certain number of LUTs that are the basic computing elements inside FPGAs. An n-input LUT is an *n*-address memory used to store the 2^n possible values of an n-inputs boolean function. With an n-input LUT it is possible to implement any function with *n* variables. The values of the function for any combination of the *n* variables is computed and stored in the LUT. The actual variables are used to address the LUT at the location where the correct value is stored. The result appears at the LUT output.
2. **Flip Flops (FF):** Flip Flops are used to temporally store values. The value to be stored in the FF can be the LUT-output or a signal with an external source.
3. **Multiplexers (MUX) :** Multiplexers are used in CLBs to connect the LUT-output or another CLB-input signal to the FF-input or to a CLB-output

Figure 2.2 shows a Xilinx Virtex II CLB [20]

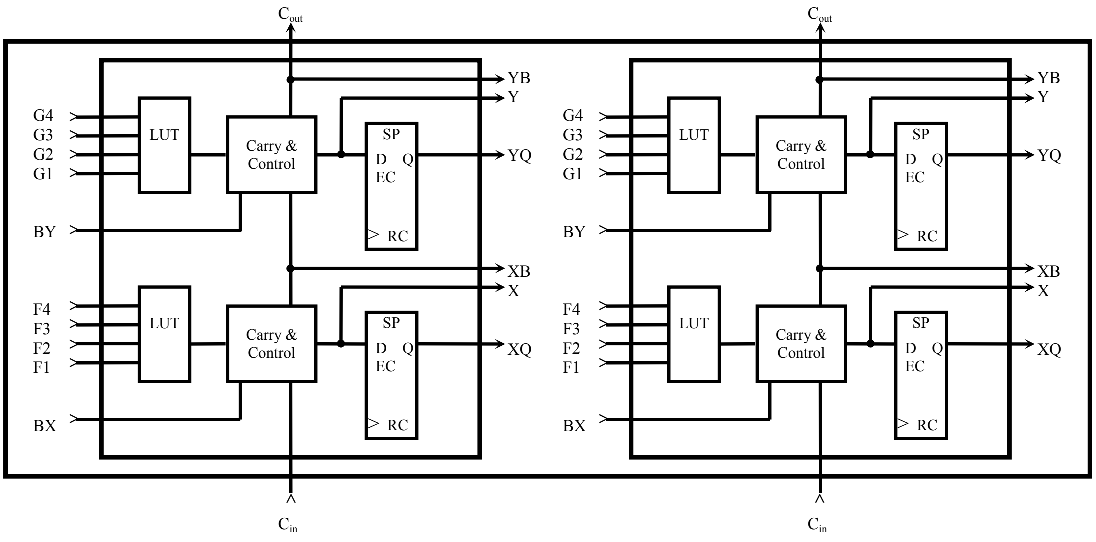


Figure 2.2 Xilinx Virtex CLB

FPGA chips can be classified into three categories according to the structure of their configurable parts [21]. These categories are Static Random Access Memory (SRAM) based FPGAs, Electrically Erasable and Programmable Read Only Memory (EEPROM) based FPGAs, and antifuse-based FPGAs. Because the configuration of an antifuse is permanent, antifuse-based FPGAs are one time programmable devices. The configuration of an EEPROM-based FPGA can be changed electrically using high-voltage electrical signals. SRAM-based FPGA chips can be reconfigured by loading the bits in the configuration file into the SRAM memory cells. These chips can be reconfigured at run time by loading a new configuration to the SRAM cells. Since they use the same technology as computer memories, they have to be configured each time the system is powered on. Because the ease of configuration, SRAM-based FPGA chips are the most widely used FPGA chips. Also, theoretically these chips can be reprogrammed an infinite number of times.

The design process for FPGAs is aided through the use of a variety of CAD tools. These tools allow the designer to describe the design using various specification formalisms such as VHDL [22], Verilog [23], Handel-C [24], state machines, or other proprietary languages. Using this specification, the tools perform synthesis of the design to generate a gate-level description of the system. This is followed by place and route tools which fragment the design into the FPGAs basic logic components and determines optimal interconnecting schemas.

2.2 Reconfigurable Computing

A Reconfigurable Computing (RC) system can be defined as a computer system that contains one or more general purpose processors and one or more configurable hardware components that are designed for their hardware functionality to be configured by the user for different applications [25]. Usually the general purpose processor acts as the host processor and the reconfigurable hardware components are used as a coprocessor. The users of the RC system typically partition their applications and execute the computationally complex sections on the reconfigurable hardware to potentially increase performance.

Reconfigurable computing exploits configurable computing devices, such as FPGAs, so they can be customized to solve a specific application [7]. Due to its potential to accelerate a wide variety of applications, reconfigurable computing has become the subject of a great deal of research. Its key feature is the ability to perform computations in hardware to improve performance, while maintaining the flexibility of a software solution. This flexibility is significant as it reduces the costs in comparison to an ASIC when changes to the system are required.

Many types of programmable logic are available. The current range of offerings includes everything from small devices capable of implementing only a handful of logic equations to huge FPGAs that can hold an entire processor core. In addition to this difference in size there is also much variation in architecture.

At the low end of the spectrum are the original Programmable Logic Devices (PLDs). These were the first chips that could be used to implement a flexible digital logic design in hardware. Other names that might be found for this class of devices are Programmable Logic Array (PLA), Programmable Array Logic (PAL), and Generic Array Logic (GAL). As chip densities increased, the PLD manufacturers evolved their products into larger parts called Complex Programmable Logic Devices (CPLDs). For most practical purposes, CPLDs can be thought of as multiple PLDs in a single chip. The larger size of a CPLD allows the implementation of more complicated designs.

At the high-end, in terms of numbers of gates, FPGAs can be found. These devices provide an array of reconfigurable logic resources consisting of combinational logic functions, flip-flops, and programmable interconnections. Functionality of current FPGA devices is established by programming.

FPGAs have a different architecture than PLDs and CPLDs, and typically offer higher capacities. The primary differences between CPLDs and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnections. This makes them far more flexible, in terms of the range of designs that are practical for implementation within them, but also far more complex to design for.

Another notable difference between CPLDs and FPGAs is the presence in most FPGAs of higher level embedded functions, such as adders and multipliers, and embedded memories. A related, important difference is that many modern FPGAs support full or partial in-system reconfiguration, allowing their designs to be changed “on the fly” either for system upgrades or for dynamic reconfiguration as a normal part of system operation. Some FPGAs have the capability of partial reconfiguration that allows one portion of the device to be reprogrammed while other portions continue running.

FPGAs were originally created to serve as a hybrid device between PALs and Mask-Programmable Gate Arrays (MPGAs). Like PALs, FPGAs are fully electrically programmable, meaning that the physical design costs are amortized over multiple application circuit implementations, and the hardware can be customized nearly instantaneously. Like MPGAs, they can implement very complex computations on a single chip, with devices currently in production containing the equivalent of over a million gates.

FPGAs have an order of magnitude more raw computational power per unit of area than conventional processors. For many applications, FPGA implementations of algorithms offer substantial improvements in energy consumption and execution speed [26] over conventional microprocessor implementations. FPGAs can complete more work per unit of time for two key reasons, both enabled by the computation's spatial organization:

- With less instruction overhead, the FPGA packs more active computations onto the same silicon die area as the processor; thus, the FPGA has the opportunity to exploit more parallelism per cycle.
- FPGAs can control operations at the bit level, but processors can control their operators only at the word level. As a result, processors often waste a portion of their computational capacity when operating on narrow-width data.

Because of these features, FPGAs had been primarily viewed as glue logic replacement and rapid-prototyping vehicles. However, the flexibility, capacity, and performance of these devices have opened up new opportunities in high-performance computation, forming the basis of reconfigurable computing.

In Figure 2.3 it is shown the tradeoff between flexibility and efficiency as well as the position of reconfigurable devices compared to processors and ASICs.

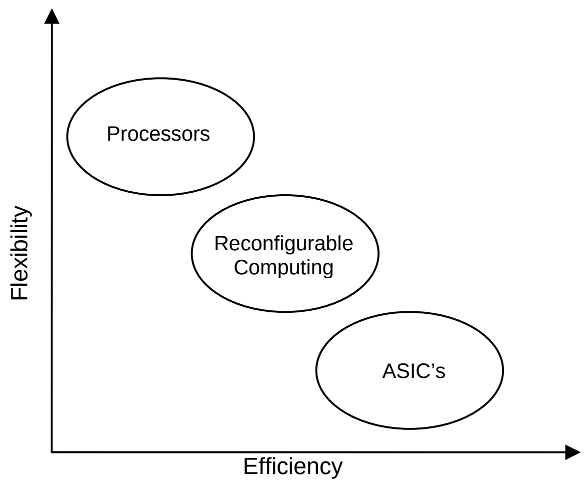


Figure 2.3 Computing paradigms

Processors have a general, fixed architecture that allows tasks to be implemented by temporally composing atomic operations, which are provided for example by an Arithmetic Logic Unit (ALU) or a floating-point unit. In contrast, ASICs implement tasks by spatially composing operations, which are provided by dedicated computational units like adders or multipliers. Reconfigurable computing combines these computing paradigms by means of reconfigurable hardware structures, which allow tasks to be implemented both in time and in space providing a higher level of flexibility.

In the GPP case, the instructions (composed into the program code) define the behavior of the computing element. The behavior of a reconfigurable device is specified by its configuration. The behavior of an ASIC is typically hard-wired and does not allow for any dynamic adaptation, except maybe for some adjustable coefficients.

It has been shown that executing computationally complex sections of applications on RC systems significantly reduces the execution time of the applications compared to the general purpose processor only systems [27]. However, applications must be mapped to FPGA devices before they can be executed on these systems. The mapping processes can be performed either manually or automatically using software tools. Several applications have been mapped to RC systems including image processing algorithms, genetic optimization algorithms, and pattern recognition.

Configurable computing generally presents four main benefits:

1. **Performance**. FPGA enable custom computing systems to be highly specialized to specific data, as well as specific applications. One optimization technique uses a form of partial evaluation, where some of the data are assumed static; the FPGA circuit is optimized to take advantage of this static data. Another typical optimization is to implement highly parallel architectures that can exploit significant data-level parallelism. By capitalizing on these optimization opportunities, a highly-tuned, yet programmable, system can be constructed that attains near-ASIC performance.
2. **Cost Effectiveness**. Configurable computing can be used to reduce system costs. A number of research efforts have demonstrated time-shared methods for simulating a large circuit on a smaller FPGA [28-30]. Furthermore, as the feature size of semiconductor processes shrink, silicon foundries are raising mask charges and minimum fabrication runs. These trends are driving low volume digital designs away from state-of-the-art process technology, and making FPGAs much more cost effective.
3. **Custom I/O**. FPGAs provide an extremely rich and flexible set of programmable I/O signals. These components give the system designer an opportunity to reuse existing hardware, or commit earlier to a new hardware

design. The benefits of custom I/O extend beyond the prototype stage, if system functionality changes after deployment the ability to rewire system I/O through reprogramming, the FPGA can be an invaluable advantage.

4. **Shorter time to market.** Products that eventually target ASIC platforms can be released earlier using reconfigurable hardware. In many market segments, the early market entry compensates for the more expensive and power hungry nature of the initial product series.

2.2.1 Types of Reconfigurable Computing

There are two types of reconfigurable computing that are characterized based on the manner in which they utilize the reconfigurable computing device [31]. The first type, which is most broadly used, is *Compile Time Reconfiguration* [32]. This is when the configuration of the computing device is decided at compile time. In this environment, the reconfigurable device is programmed at the beginning of execution, and remains unchanged until the application has finished. The second type of reconfigurable computing is *Runtime Reconfiguration* (Dynamic Reconfiguration). In this computing paradigm, the application consists of a set of tasks that can be downloaded into the reconfigurable device [32]. During the execution life of the application, the reconfigurable device is re-programmed a number of times from a set of tasks.

2.2.2 Dynamic Reconfiguration

FPGA reconfiguration typically requires an entire device to be reconfigured even for the smallest circuit change, thus interrupting all other circuits operating on the array during this period. Ideally, it is desired to configure only the area that is being updated, without interrupting the rest of the system operation. This type of reconfiguration is referred to as Dynamic Reconfiguration or run-time reconfiguration and has been actively researched for the last decade [33, 34].

In the area of reconfigurable computing, dynamic reconfiguration has emerged as a particularly attractive technique to increase the effective use of programmable logic blocks. Dynamically Reconfigurable Logic (DRL) devices allow the change of the device configuration on the fly during system operation.

FPGAs that are not dynamically reconfigurable must be off-line before a reconfiguration cycle, partial or full, can begin. If such a suspension is required then this limits what can be actually done efficiently on the device. Usage, for example, as a programmable algorithm accelerator, is limited in this case, as even small changes to functionality become critical to its whole operation. Certain algorithms, therefore, become undesirable in this kind of set-up. Such devices also have problems with task organization, when multiple tasks are being performed concurrently, as certain tasks are likely to complete before others, thus making it difficult to optimize the whole design for maximum speed advantage.

If a reconfigurable FPGA only supports complete reconfiguration the disadvantages multiply. As such devices increase in density and do not include the ability to partially reconfigure; this is likely to become more problematic.

Reconfiguration time is a crucial parameter for any dynamically reconfigurable computing system [35]. The reconfiguration time imposes limits on the applications that can be efficiently mapped to the reconfigurable hardware. Reconfigurable devices, in particular FPGAs, show relatively long reconfiguration times in the range of dozens of milliseconds [36]. Consequently, rather long-running application functionality is mapped to such devices. For shorter term functionality the reconfiguration overhead can be significant, negating any performance gain over software. The trend towards ever larger devices intensifies the problem further, because the reconfiguration time is proportional to the amount of configuration data, which grows with the device size.

The data required to configure a reconfigurable device is commonly denoted as a context [37]. Depending on the capabilities of the device, two basic classes are distinguished. Single-context devices store exactly one configuration on the chip. Before a new context can execute, the corresponding configuration data has to be loaded from off-chip. Conventional FPGAs fall into this category. Multi-context devices hold a set of configurations on-chip. At any given time exactly one configuration is in use, the so-called active context. To execute a new context, the contexts are switched, i.e. a previously inactive, stored context becomes active. Since the configuration data does not have to be loaded from off-chip, the context switch is significantly speeded up.

In DRL, a circuit or system is adapted over time. This presents additional design and verification problems to those of conventional hardware design [38] that standard tools cannot cope with directly. For this reason, DRL design methods typically involve the use of a mixture of industry standard tools, along with custom tools and some handcrafting to cover the conventional tools inadequacies. The lack of commercial Computer Aided Design (CAD) tool support and complexity of designing dynamically reconfigurable hardware has led to the creation of a number of research tools such as JHDL and JBits.

JHDL [39, 40] is a free Java based tool for FPGA design. This tool is a result of BYU research. JBits [17, 41] is the only commercial available tool that supports dynamic and partial reconfiguration of Xilinx Virtex FPGAs using standard Java functions. Xilinx uses this language in its Internet Reconfigurable Logic via standard Java applications. These tools are based on Java language which is the most popular software language with Internet support.

Current tools are slow and hardware oriented and there is no way to program such systems in a general purpose manner. Other software tools are also required which suit reconfigurable systems, such as:

- Simulation Models for dynamic reconfiguration.
- Automatic design partitioning, based on temporal specifications.
- Support for generation of relocatable bitstreams.
- A simulation package for modeling new FPGA architectures.
- Debugging tools.

Reconfigurable computing is an active research area with several new directions being explored to develop better architectures, methodologies, algorithms and software tools. There is an acute need for software tools that permit simulation of dynamic reconfiguration and mapping of applications onto dynamically reconfigurable architectures.

2.3 Handel-C

Handel-C [42] is a programming language whose objective is to allow the designer a fast prototyping of applications on target FPGA from specifications in a high-level language. Handel-C uses syntax similar to conventional C language, with the addition of parallel descriptors and some features to handle channels and interfaces.

Sequential programs can be written in Handel-C just as in conventional C, but to gain the most benefit in performance from the target hardware, its inherent parallelism must be exploited [40]. As with conventional high level languages, Handel-C is designed to allow the expression of the algorithm without worrying too much about exactly how the underlying compilation engine works. This philosophy makes Handel-C a programming language rather than a hardware description language.

Although Handel-C is inherently sequential, it is possible to instruct the compiler to build hardware to execute statements in parallel. Handel-C parallelism is true parallelism, it is not the time sliced parallelism, familiar from general purpose computers. When instructed to execute two instructions in parallel, they are executed at exactly the same instant in time by two separate pieces of hardware.

The principle of Handel-C is the following: during the phase of compilation, every instruction on the source program is transformed into a material entity which makes one or several treatments on every associated instruction to generate a set of material blocks representing all the instructions of the program. The DK design suit synthesizes the design and generates a netlist in an EDIF format. The netlist is then passed to a vendor place and route tool which generates a placement and routing, and finally a bitstream to configure the device. Because of its high abstraction level, Handel-C allows coding of complex algorithms without having to consider lower-level designs. Using Handel-C constructs, the development cycle for the creation and testing of FPGA designs can be accelerated. In addition, the package includes a library of basic functions and a memory controller to access the external memory on the FPGA board.

2.4 Related Work

Parallel computing has been major subject of research since at least the early 1980s. This approach has permitted to solve complex problems and to implement high-performance applications in areas such as science and engineering.

A parallel computer is constituted of a set of processors that are able to work cooperatively to solve a computational task. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems.

Parallel computers consist of three main building blocks: processors, memory modules, and an interconnection network [43]. There has been steady development of the sophistication of each of these building blocks, but it is their arrangement that most differentiates one parallel computer from another. Parallel computers are interesting because they offer the potential to concentrate computational resources, whether processors, memory, or I/O bandwidth, on important computational problems.

The processors used in parallel computers are increasingly alike to the processors used in single-processor systems [43]. Present technology, however, makes it possible to fit more onto a chip than just a single processor, so there is considerable investigation into what components give the greatest added value if included on-chip with a processor. Some of these, such as communication interfaces, are relevant to parallel computing.

Special purpose parallel systems can be accomplished using a wide variety of technologies. The design and fabrication of a custom VLSI is one approach to implementing a special purpose system. The principles of VLSI design have become not only a mainstream part of electronics but also of computer science. Recently a large number of new and interesting VLSI designs for basic problems in computer science have been produced.

In deciding whether to implement a special purpose parallel system as a custom VLSI chip, one have to weigh the advantages against the disadvantages. The arguments in favor of a custom VLSI implementation are typically that it is feasible to obtain the highest possible performance by that means and that the custom VLSI system can be manufactured more cheaply. The latter may be very significant if the chip forms part of a high volume production. The major disadvantages of custom VLSI design and implementation is that it is a very costly and time consuming process, and that it results in a product which can not easily be changed. Although a large number of interesting and detailed VLSI algorithm designs have been developed over the last decades, remarkably few of them have actually been accomplished as custom chips. The reason for this is that few organizations can justify the prohibitive cost of producing such a chip.

Most VLSI systems designed today are monolithic and cannot easily be modified or improved [44]. A significant factor giving rise to this design difficulty and inflexibility is the synchronous nature of such systems. In recent years, a number of researchers have been investigating the possibilities of designing alternative forms of VLSI systems to overcome the limitations imposed by the traditional implementations. Such approaches offer the opportunity to build up complex systems by hierarchical composition of simpler ones. The resulting system designs are easier to modify and improve, in order to meet changing requirements or to take advantage of developments in VLSI technology.

VLSI technology provides an effective approach for high performance image processing applications [45]. Since data parallelism can be taken into account fully in the system design, VLSI processor arrays have the potentiality for deep parallel processing.

Due to the regular structure and simple control strategy, VLSI processor array can achieve an excellent cost-performance ratio. In some special applications, VLSI array is the effective way to decrease the size, the power consumption, the heat dissipation, and improve the reliability.

VLSI processing architectures have been highly influenced by the systolic array approach. A systolic processor is made up of a series of interconnected cells, each one of which is designed to carry out operations rhythmically, incrementally, cellularly, and repetitively [46]. As H. T. Kung pointed out, systolic arrays design results in architectures which are simple and regular and therefore they are both modular and expandable [47].

The computation tasks can be divided into two conceptual categories: compute-bound and input/output limited [48]. Systolic array is an effective solution for compute-bound problems. Being algorithm dedicated, for many computation-intensive image processing tasks, systolic arrays can outperform general microprocessors several orders of magnitude under the same manufacture technology. However, they usually can not be upgraded or adapted to other purposes. Therefore, special purpose VLSI solutions are generally used in mature image processing application areas. In this regard, they are being rivaled by the field programmable device technology.

In the last decade several companies, like Xilinx, started to offer an attractive alternative to custom VLSI implementation based on FPGAs. Reconfigurable computing has been successfully used in many compute intensive areas, including image processing [49]. Currently, a large number of reconfigurable computing solutions are available [19]. Despite the great amount of research done on FPGAs, many FPGA-based applications have been algorithm specific. An environment for developing applications needs more than just a library of static FPGA configurations, perhaps parameterizable, since it should allow the user to experiment with alternative algorithms and to develop his own algorithms. There is a need for bridging the gap between high level application-oriented software and low level FPGA hardware. Many behavioral synthesis tools have been developed to satisfy this requirement. These tools allow the user to program FPGAs at a high level without having to deal with low level hardware details [50] (e.g. scheduling, allocation, pipelining). However, although behavioral synthesis tools have developed enormously, structural design techniques often still result in circuits that are substantially smaller and faster than those developed using only behavioral synthesis tools.

2.4.1 Image Processing Architectures Taxonomy

Architectures for image processing can be classified depending on the type of algorithm namely low-level, intermediate-level or high-level. Yet another way to classify the architectures is based on the instruction and data streams. The two common classes are Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) [51]. Typically architectures for low-level image processing tend to be SIMD or systolic arrays because parallelism is more obvious compared to high-level algorithms. Table 2.1 adapts a taxonomy presented by Maresca in [52] based on Flynn's taxonomy with some system examples based on FPGAs.

Some common structures in SIMD processors for low level image processing systems are Meshes, array processors either 1D or 2D [53], or systolic arrays of processing elements [54], hypercubes and pyramids. Examples of mesh connected computers include CLIP4 [55] and the MPP [56]. The Connection Machine [57] uses a hypercube network for long distance communication. PAPIA [58], SPHINX [59], MPP pyramid and HCL Pyramid are examples of pyramid structure.

Table 2.1 Image Processing Architecture Taxonomy

Flynn's	Type of Autonomy		Network Topology	Data Width	Examples	FPGA-based
MIMD			Multi-stage	Floating Point	PASM [71]	Armstrong III [90]
					IUA [72]	
			Hypercube	Floating Point	NCUBE [73]	
					iPSC [74]	
			Bus oriented	Mixed	DATA CUBE [75]	Splash2 [91]
			Linear	Floating Point	WARP [68]	Rapid [92]
			Ring	Integer	ZMOB [76]	
			Mesh	Floating Point	Transputer based [77]	
MISD			Linear	Bit-Serial	Cytocomputer [78]	Wiart [93]
				Integer	PIPE [79]	
SIMD	Non-Autonomous		Mesh	Bit-Serial	CLIP4 [54]	
					MPP [80]	
					DAP [81]	Perry [94]
					AAP [82]	
					GRID [83]	
			Tree	Integer (8 bits)	NON VON [84]	
			3-D Cube	Bit-Serial	Hughes Wafer Snack [85]	
	Autonomous	Addressing	Mesh	Integer (8, 16, 32 Bits)	ILLIAC4 [86]	MGAP [95]
			Linear	Integer (16 Bits)	CLIP7 [87]	
			Multi-Stage	Floating Point	GF11 [88]	
					PASM [71]	
		Connection Autonomy	n-Cube	Bit Serial	Connection Machina [57]	
			Polymorphic Torus	Bit Serial	YUPPIE [89]	VIP [96]
		Operation Autonomy	Pyramid	Bit Serial	PAPIA [57]	
			Linear	Integer	CLIP7 [86]	

Various systems based on systolic arrays have been presented, RAW [60], PipeRench [61] and TRIPS [62], which are focused on exploiting spatial computation. In [63-66], special purpose convolution architectures are presented; they are designed to meet real-time image processing requirements. Hsieh and Kim in [63] proposed a highly pipelined VLSI convolution architecture. Their approach considers parallel one dimension convolution and a circular processing module to achieve high performance using an array of $n \times n$ processing elements, each being a multiplier and adder.

In [64] Haule and Malowany and Hecht et al in [66] proposed convolution architectures based on systolic arrays which operate on real time images with a size of 512×512 pixels, both architectures perform bit-serial arithmetic. The architecture of [63] requires on chip memory to store the necessary input pixels.

The architectures presented in [63] and Hsu et al in [65] are special purpose architectures for convolution, and both of them require $n \times n$ processing elements which could potentially occupy a large chip area. The architectures of [64] and [66] are both systolic array architectures employing bit-serial arithmetic operations and hence may not be able to meet the performance requirements mentioned above. In [66] the authors point out the well known fact that for most applications bit-parallel arithmetic has a performance edge over bit-serial arithmetic. However, the processing architecture of [67] is based on bit-serial arithmetic since it is sufficient for their requirements and has lower gate count.

Another approach involves language and compiler design to capture parallelism. Algorithms are written in a high-level language and a specialized compiler is required to map computation to the systolic array. Several examples include the Warp processor [68], which is a systolic array machine proposed and built for image understanding and scientific computations. The machine has a programmable systolic array of linear connected cells. iWarp [69], the next version of Warp, is a two dimensional systolic and distributed memory architecture considered for image understanding and scientific computations. iWarp supports memory communications and systolic communications.

Chai and Will [70] extended the collection of research investigating I/O interconnection in a systolic array to exploit the physical data locality of planar streams by processing data where it falls to improve performance.

Table 2.1 summarizes several VLSI and FPGA-based architectures, most of them consist of special-purpose systolic arrays with a high-performance data-flow structures which are characterized by having simple and regular communication. However, the principal drawback to these special-purpose arrays is that they are designed for specific algorithms which possess very simple and regular data flow patterns.

Besides the aforementioned, when applied to the computer vision field the architectures presented in Table 2.1 present some unsolved problems:

- The systems are not optimized for vision applications, or they are just specific for a given image algorithm.
- The architectures are not focused in solving high-level vision tasks.
- The systems are designed without considering modularity.
- Systems require complex schemas either for communication with the main processors or for data interchange and management.
- Complex applications require simple operators most of which involve local data calculations. Regularly images are transmitted from the host to the coprocessor and back to the host for each operation, data transmission times then overwhelm any benefit from parallel calculation.
- The amount of data managed is a disadvantage for some applications implemented. For large images, the communication bandwidth between the

host and RC processors becomes the limiting factor, implying that data traffic must be minimized.

- Most systems have high power consumption requirements.

These restrictions limit the type of algorithms or applications that can effectively be supported by such architectures under real time. The present work explores the possibilities for implementing efficiently computer vision tasks using FPGA technology overcoming such drawbacks.

The proposed architecture is based on a systolic array belonging to an Enhanced version of a Systolic Array (ESA). The primary objective of this research is to develop and to evaluate an architecture that gives support to window-based image operators, processing at least 30 frames per second using 640×480 gray level images.

The purpose of the architecture is to implement a particular kind of algorithms known as window-operators such as filtering, erosion, dilation, Gaussian pyramid, template matching and matrix by matrix multiplication. However the architecture broadens the scope of algorithms executable on systolic arrays while retaining much of the simplicity and regularity of the original systolic array architecture.

The architecture has been provided with elements that allow flexible routing of data and storage elements that allow reusing information to chain processes. Motion estimation has been implemented to test this feature. Based on a chaining schema, the system allows the implementation of higher level vision tasks resulting in a non-dedicated, scalable and, modular architecture.

The main objective for the implementation of the architecture is the flexibility. Flexibility in the sense of modularity refers to the capability of having a systolic structure that allows parts to be added, removed, replaced, or modified. In this case PEs functions can be modified to perform different operations according to a control word chosen by the user depending on the algorithm applied. The number of processing elements to be active in a current application can be defined also by the user at start up. Scalability is implemented by extending the generic design of the systolic array architecture to allow configuring the architecture by setting generic parameters according to specific needs to meet different performance requirements or to add functionality to meet the changeable demands of the applications.

Most of the architectures that have been developed use off-chip memory banks to store the large amounts of data managed by implemented algorithms. Memory architecture organization then becomes critical for an optimized design since frequent memory accesses can result in long delays and degrade the system performance. The proposed memory access structure uses local buffers that contribute to form an expandable and scalable architecture and enhances the performance, silicon efficiency and scalability of the architecture.

The capability of partially buffer an image, ordering the pixels in the way that is required for the application helps to deal with bandwidth constraints and adds flexibility in the kind of algorithms that can be handled.

The logic blocks themselves can be configured to act like RAM but this is usually an inefficient managing of hardware resources. Using off-chip memory for buffering may only allow a single access to data per clock cycle. The proposed buffering schema allows parallel access to the data handled by systolic processor and an optimized use of silicon. Furthermore the memory buffers provide a means to re-use information among several processors cascade connected.

Input and Output buffers are used to transfer data to or from external DRAM while active data are processed by the FPGA architecture achieving a minimal overhead. Computing elements and memory buffers are mixed in regular patterns to form a high level pipeline schema which supports different algorithms with similar data requirements. This proposal opens the possibility of a very complex multi-function image processor with essentially the same per-unit hardware cost as a single function system.

In the next chapters, detail description and evaluation of the proposed architecture are presented. Additionally the advantages and shortcomings of the architecture are outlined.

Chapter 3

Systolic Arrays

Since the early days of parallel computing a variety of parallel algorithms has been described that can be classified as systolic [46]. The terminology systolic has been cast in the early eighties by H. T. Kung and C. E. Leiserson [1]. In a series of seminal publications they have established systolic algorithms as a novel class of parallel computing structures [1, 97].

At that time, advances in semiconductor technology have lead to the development of VLSI chips that, for the first time, allowed the implementation of basic algorithms directly in hardware [98]. As such VLSI circuits could be customized for specific applications; it could be expected that the combination of dedicated hardware and specific algorithm, with increasing integration, would lead to high parallelism. However, for most systolic algorithms described, the level of integration, i.e., the number of systolic processing elements which could be etched on a chip, did not suffice at that time for an efficient VLSI hardware implementation.

The successful application of general purpose parallel computers for science and engineering, starting in the early nineties, has then led to a rebirth of systolic algorithms which are readily mapped and well adapted to parallel machines, their ideal implementation medium. Its advantages have been recognized by many designers and producers of massively parallel computers at a rather early stage [99-101].

3.1 Parallel Architectures for Image Processing

Modern image processing and computer vision algorithms require high computational capability especially when high resolution images have to be elaborated under real-time requirements [102]. In such applications (e.g. image filtering, image restoration, feature recognition, object tracking, template matching, etc.) a rate of millions of bytes per second must be processed. These characteristics make image processing suitable for parallel processing.

Image processing involves many types of computing, ranging from two-dimensional correlation and convolution, to image transformation, geometric computing and graph analysis [52]. Each of these types of computation places specific requirements on image processing systems although a very general characteristic is the execution of a large number of computations on regular data structures.

In many cases the execution of image processing tasks consist of the application of a set of operations to all the elements of the image array, and this can be done concurrently on each element of the array, allowing significant speedup through parallelism [52]. While this is obviously true for image processing, it is also true in many other vision tasks, even though sometimes parallelism can be exploited only by clever algorithms.

Depending on the processing, the involved data structures and the communication needs, an image processing application can be divided into three levels: *low*, *intermediate* and *high* [103, 104].

Low level image processing is normally termed bottom-up processing. Most image-processing operations fall into this category. Input data includes images or simple transformations of images. These operations primarily work on whole image data structures, and yield another image data structure.

Computations in low-level processing are to be performed for each pixel in an image, they are regular, exhibit high spatial parallelism, and mainly involve numeric processing. The steps are simple and repetitive, the number of these steps is constant and the data exchange, handled in a static way, is locally done. These computations are well suited for both SIMD and MIMD architectures. Example algorithms include edge detection, filtering operations, and connected component labeling [19], [105-110].

Intermediate level image processing conveniently bridges bottom-up (low-level) and top-down (high-level) processing. These operations reduce the image data field into segments (regions of interest), and produce more compact and symbolic image structure representations (such as lists of object border descriptions).

Processing on this level attempts to build a coalition of tokens to extract meaningful entities, for example, forming rectangles from lines. Computations in this category manipulate symbolic and numeric data. They are normally irregular and data dependent. The steps are recursive, iterative, and their number is changeable. Data exchanges have an uneven rate; their type is global and depends upon the image content. They are suitable for medium, to coarse, grain parallelism in MIMD mode, although a subset of computations also can be performed efficiently on SIMD architectures.

High level image processing tasks are normally top-down (or model-directed), they are more decision-oriented and they primarily concern the interpretation of the symbolic data structures obtained from the intermediate level operations. Essentially, the operations try to imitate human cognition and decision making, according to the information contained in the image. Examples are object recognition, and semantic scene interpretation [111-114].

Processing at this level is not as well defined as on the other two levels. Furthermore, processing in this domain may require re-executing algorithms from the other two levels. Although parallelism on computations at this level is not well understood, it is believed necessary to dynamically schedule computations. The diversity and highly data-dependent nature of computations make this level of processing largely suitable for MIMD parallelism.

Parallel hardware architectures can effectively speed up a computation system to reach a performance level that is higher than that of a single processor. Nevertheless, mapping algorithms in hardware is in general a difficult task. When parallelism is introduced in the execution of an application it must fit the target architecture [115], which sometimes is constrained by the available technology. When parallel processing is applied, four main level of parallelism can be distinguished: *job-level*, *task-level*, *instruction-level*, and *gate-level* [116].

Parallelism at job-level. A parallel computer is a computer having more than one processor executing a single application simultaneously [43]. Supercomputers are the most expensive and most powerful category of parallel computers [117]. They are typically used for scientific and engineering applications that must handle very large databases or do a great amount of computation like in image processing and computer graphics applications [118]. One of the leaders in visualization supercomputers manufactures is Linux Networx, the Linux Supercomputing Company. They develop powerful visualization systems for many technical fields such as: modern science with data-intensive requirements security and defense, modern design processes and advanced media applications.

Parallelism at task-level. Parallel processing does not refer only to many processors working in parallel; software can do this, as well. In parallel programming there are many programs processing the data in different ways at the same time. Thus, multiple programs can be executed in parallel, if there are no data dependencies. In the case of data dependency, several parallel programming models are commonly used: Shared Memory, Threads, Message Passing, Data Parallel or Hybrid [119]. Many software libraries for parallel image processing are currently available. They are a set of routines which allow the user to perform operations on images in a parallel manner. Many implementations of well known programming language compilers, such as C and C++, are used today for compiling high-performance computing applications based on parallel paradigms.

Parallelism at instruction-level. Parallelism at instruction-level can be defined as a measurement of the number of operations that can be performed simultaneously in a computer program. Superscalar architecture refers to the use of multiple execution units, so that more than one instruction can be processed at a time [120]. These multiple parallel processing units are inside one processor. Most of the modern processors are superscalar [121, 122]. When the instructions are pipelined into the processor, there is a mechanism which selects the instructions which will be executed in parallel. Some compilers can make easier this process by sending the instructions in a proper way to the processor. Another way to achieve parallelism at instructions level can be seen in the case of vector processors. They provide high-level operations that work on vectors (linear or array) instead of a single data. A vector instruction is equivalent to an entire loop instruction. This type of processing avoids data hazards, reduces instruction bandwidth requirements and is obviously faster than scalar operations. In general DSP devices have hardware which supports the execution of vectors operations. Advanced DSP processors integrate instruction parallelism where several RISC-equivalent (Reduced Instruction Set Computer) operations can be executed in parallel. These architectures allow the individual machine operations to overlap (addition, multiplication, load, and store). Many of these fully programmable processors are used in multimedia applications as well as image coding and decoding (TMS 320 C6xx series) [123, 124].

Parallelism at gate-level. Hardware devices based on array architecture are able to execute a large amount of logic operations at the same time. In parallelism at gate-level, bit-level operations are executed in parallel. Moreover, the new devices in this category provide the necessary computing resources to meet the high-performance required in digital signal processing applications. An important advantage of gate-level parallelism is that the designer can implement as many parallel resources inside the device as necessary to achieve the performance required by the system. The resources are not fixed like in general purpose processors where each processor contains a finite number of basic computing functions. This thesis exploits the characteristics of gate-level implementation to test parallel architectures for image processing with the aim of obtaining video rate performance.

In spite of the large potential performance of parallel architectures, the image processing community does not benefit a hundred percent for high-performance computing. Essentially, this is due to the lack of optimized programming tools that can effectively help non-expert parallel programmers to develop multimedia applications for high-performance parallel architectures. The main objective of parallel processing is to wipe out the physical limits of serial processors by employing several processors working in parallel in order to reduce the execution time. Thus, a huge amount of research has been carried out in the field of parallel processing in past years, either concerning parallel hardware architectures or algorithms and programming languages [125-129].

3.2 Parallel Computing Models and Systolic Arrays

A well known taxonomy of parallel systems is due to Flynn. Flynn's taxonomy classifies architectures on the presence of single or multiple streams of instructions and data [130, 131]. This yields the four categories below. See Figure 3.1 to 3.4:

- **SISD** (Single Instruction, Single Data stream) - defines serial computers.

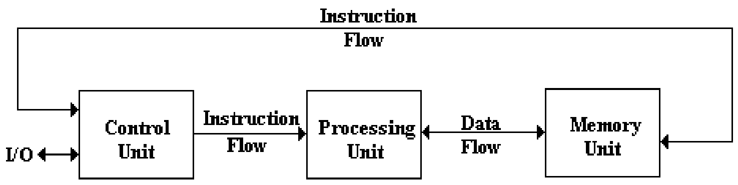


Figure 3.1 SISD architecture in Flynn's Taxonomy

- **MISD** (Multiple Instruction, Single Data stream) - would involve multiple processors applying different instructions to a single datum; this hypothetical possibility is generally deemed impractical.

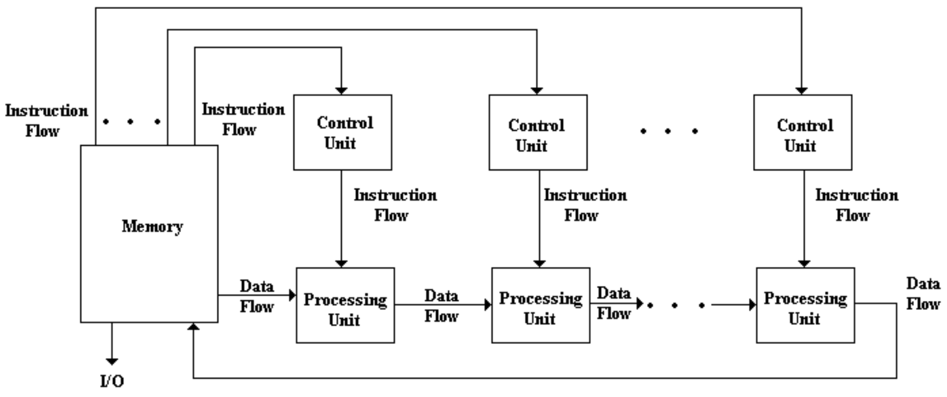


Figure 3.2 MISD architecture in Flynn's Taxonomy

- **SIMD** (Single Instruction, Multiple Data streams) - involves multiple processors simultaneously executing the same instruction on different data

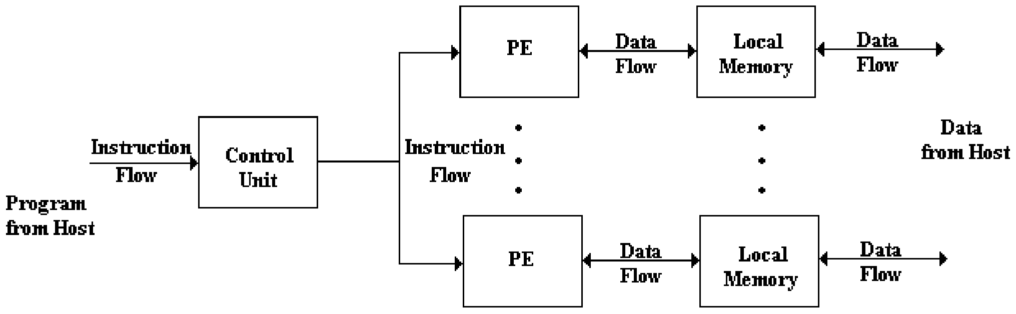


Figure 3.3 SIMD architecture in Flynn's Taxonomy

- **MIMD** (Multiple Instruction, Multiple Data streams) - involves multiple processors autonomously executing diverse instructions on diverse data.

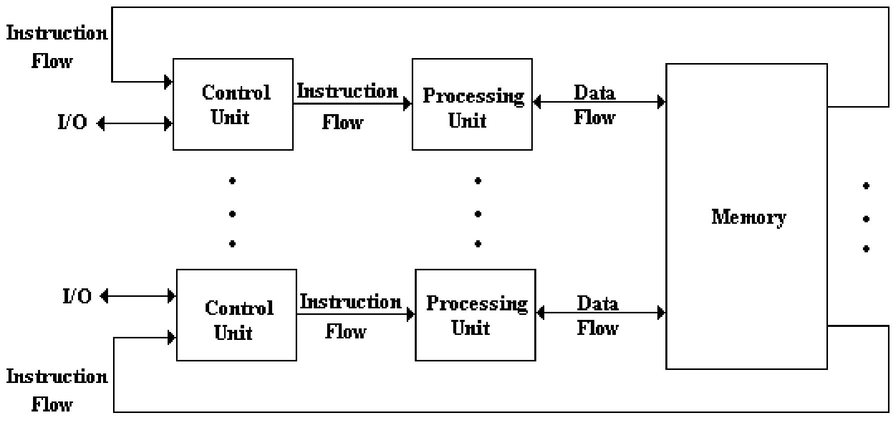


Figure 3.4 MIMD architecture in Flynn's Taxonomy

While systolic arrays were originally used for fixed or special purpose architecture, the systolic array concept has been extended to general-purpose SIMD and MIMD architectures.

Regarding general-purpose systolic arrays the two basic types are the programmable model and the reconfigurable model [132].

In the programmable model, cell architectures and array architectures remain the same from application to application. However, a program controls data operations in the cells and data routing through the array. All communication paths and functional units are fixed, and the program determines when and which paths are used.

In the reconfigurable model, cell architectures as well as array architectures change from one application to another. The architecture for each application appears as a special purpose array. The primary means of implementing the reconfigurable model is FPGA technology.

A programmable systolic architecture is a collection of interconnected, general-purpose systolic cells, each of which is either programmable or reconfigurable. Programmable systolic cells are flexible processing elements specially designed to meet the computational and I/0 requirements of systolic arrays. Programmable systolic architectures can be classified according to their cell interconnection topologies as fixed or programmable.

Fixed cell interconnections limit a given topology to some subset of all possible algorithms. That topology can emulate other topologies by means of the proper mapping transformation, but reduced performance is often a consequence [132].

Programmable cell interconnection topologies typically consist of programmable cells embedded in a switch lattice that allows the array to assume many different topologies. Programmable topologies are either static or dynamic. Static topologies can be altered between applications, and dynamic topologies can be altered within an application. Static programmable topologies can be implemented with much less complexity than dynamic programmable topologies.

Reconfigurable systolic architectures capitalize on FPGA technology, which allows the user to configure a low-level logic circuit for each cell. Reconfigurable arrays also have either fixed or reconfigurable cell interconnections. The user reconfigures an array’s topology by means of a switch lattice. Any general-purpose array that is not conventionally programmable is usually considered reconfigurable.

Programmable systolic arrays are programmable either at a high level or a low level. At either level, programmable arrays can be categorized as either SIMD or MIMD machines [132]. High-level programmable arrays usually are programmed in high-level languages and are word oriented. Low-level arrays are programmed in low-level languages and are bit oriented.

FPGA technology has produced a low-level, reconfigurable systolic array architecture that bridges the gap between special-purpose arrays and the more versatile, programmable general-purpose arrays. Purely reconfigurable architectures are fine-grain, low-level devices best suited for logical or bit manipulations.

SIMD systolic machines operate similarly to a vector processor, typically they employ a central control unit, multiple processors, and an interconnection network for either processor-to-processor or processor-to-memory communications [133]. The control unit broadcasts a single instruction to all processors, which execute the instruction in lockstep fashion on local data. The interconnection network allows instruction results calculated at one processor to be communicated to another processor for use as operands in a subsequent instruction. Individual processors may be allowed to disable the current instruction. Adjacent PEs may share memory, but generally no memory is shared by the entire array. After exiting the array data is collected in an external buffer memory.

MIMD systolic machines operate similarly to homogeneous von Neumann multiprocessor machines; they employ multiple processors that can execute independent instruction streams, using local data. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner. Although software processes executing on MIMD architectures are synchronized by passing messages through an interconnection network or by accessing data in shared memory units, MIMD architectures are asynchronous computers, characterized by decentralized hardware control.

The impetus for developing MIMD architectures can be ascribed to several interrelated factors. MIMD computers support higher level of parallelism (subprogram and task levels) that can be exploited by “divide and conquer” algorithms organized as largely independent sub-calculations. MIMD architectures may provide an alternative depending on further implementation refinements in pipelined vector computers to provide the significant performance increases needed to make some scientific applications tractable. Finally, the cost-effectiveness of *n*-processor systems over *n* single-processor systems encourages MIMD experimentation.

Each PE’s architecture is somewhat similar to the conventional von Neumann architecture: It contains a control unit, an ALU, and local memory. MIMD systolic PEs have more local memory than their SIMD counterparts to support the von Neumann-style organization. Some may have a small amount of global memory, but generally no memory is shared by all the PEs. Whenever data is to be shared by processors, it must be passed to the next PE. Thus, data availability becomes a very important issue. High-level MIMD systolic PEs are very complicated, and usually only one fits on a single integrated chip.

3.3 Systolic Systems, Arrays, and Algorithms

A systolic system is a specific hardware network, which refers to computing structures with cellular organization and pipelined data flow.

In physiology, the term *systolic* describes the contraction (systole) of the heart, which regularly sends blood to all cells of body through the arteries, veins, and capillaries. Analogously, systolic computer processes perform operations in a rhythmic, incremental, cellular, and repetitive manner.

The basic element of a systolic system is the PE. A processing element is built by a functional processing unit which can perform data manipulations and using a delay or memory element allows a controlled data flow into and out of the processing element. A processing element can contain registers which function as memory locations for storage of results of intermediate computations and for static data to be used in the course of the computation.

A collection of PEs is denoted as systolic array. A systolic array is a parallel computing device for a specific application. The PEs of the array are interconnected in a regular pattern with a limited number of neighboring elements [134]. In a similar way as blood circulates in the human body, data circulates inside the PEs of the systolic array, and interact with other data; hence a systolic array exhibits a simple regular design. The regularity and simplicity constitute a desirable characteristic for their direct implementation in silicon, in the form of VLSI chips. Figures 3.5 to 3.8 illustrate some generic examples of simple interconnection patterns of systolic arrays, linear, 1-dimentional ring, 2-dimentional squared array and 2-dimentional hexagonal array.

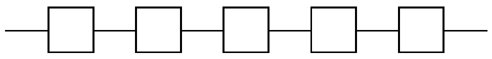


Figure 3.5 Linear systolic array interconnection

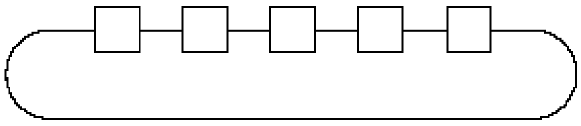


Figure 3.6 1-dimentionalring interconnection

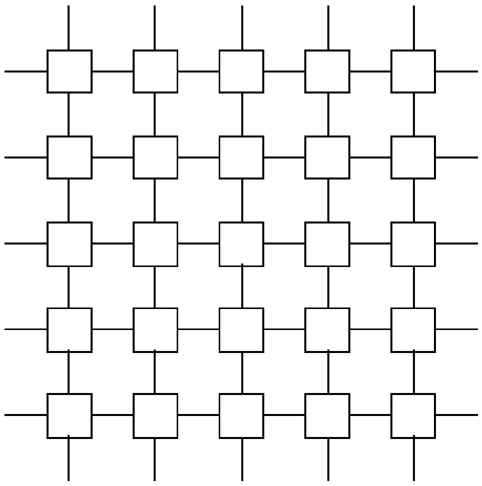


Figure 3.7 2-dimensional square array

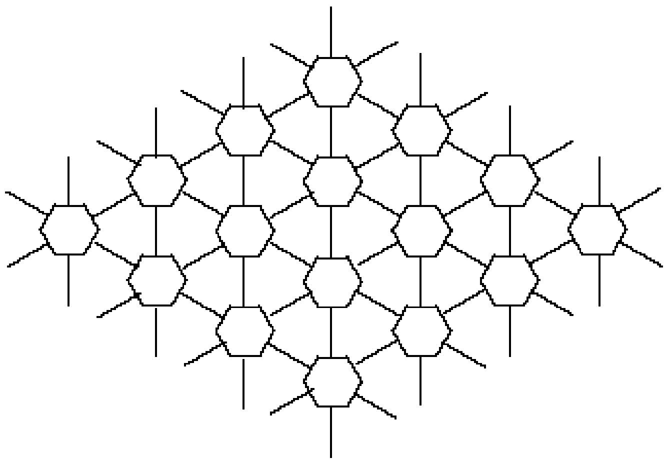


Figure 3.8 2-dimensional hexagonal array

Algorithms implemented on the systolic array are referred to as systolic algorithms. In the systolic algorithm, each of the PEs performs only a part of the necessary computations thus contributing to the totality of computations as successively carried out by the systolic array. The operations are performed by the PEs of a systolic array in a synchronous mode. The basic step of a systolic algorithm, in general, is a computation followed by the exchange of data to the nearest-neighbor PEs. A typical sequence of operations is the repeated parallel computation of identical functions between consecutive parallel communication events. In the course of the execution of the systolic algorithm, the data flow with a constant speed through the PEs of the system. The computation step sometimes is denoted as *pulse*, the communication step as *move*. This underlines the resemblance of the systolic algorithm as implemented on a systolic system with the blood circuit called systole in physiology, where the heart beat in the contraction phase (pulse) presses the blood through the blood vessels (move) with a permanent speed.

Putting together many of the pulse-and-move operations eventually, the systolic array has completed the computational problem posed, after a well defined number of usually equally spaced time steps. The time step is called systolic cycle, in general it is distinguished from the clock step of the underlying implementation system. In a systolic array, once data is available, it is used effectively inside many PEs to yield a high throughput rate. Thus, a systolic array may exploit the inherent parallelism of some applications. Under a systolic approach, all the operations are performed in a synchronous manner independent of the processed data. The only control data that is broadcasted to the processing elements is the clock.

As a summary some typical features and structures of systolic arrays are shown here:

1. A systolic array consists typically of a large number of PEs which are of the same type. If more than one type is involved, then there are only a few different processing elements.
2. The PEs execute repeatedly one and the same function. It is however possible that:
    a) Different PEs of the same type perform different operations at the same time.
    b) One PE performs different operations in different clock periods.
3. The PEs can possess registers that function as intermediate storage locations.
4. The PEs are arranged and interconnected in a regular pattern.
5. The interconnections are local, in that only neighboring PEs can communicate directly.
6. Input data is fed into the array and output data is retrieved from the array only at boundary PEs.
7. Many PEs work in parallel on different parts of one and the same computational problem.
8. The efficiency of the algorithm or equivalently the utilization of the processing resources of the array, measured in the amount of PEs which are busy with computation, is high.
9. All data is processed and transferred by pipelining.
10. Several data flows move at constant speed through the array and interact with each other during this movement.
11. Once input into the array, the data is used many times.
12. There is either no need for control or the control is very simple. The correct operation of the structure is provided by the coordinated input of the data flows. If control is needed, it is carried out by an instruction flow which moves at a constant speed through the array, coordinated with the data flow.

3.4 Window-based Operators

Image processing is an ideal candidate for specialized architectures because of the massive volume of data, the natural parallelism of its algorithms and the high demand of image processing solutions.

The rectangular structure of an image intuitively suggests that an image processing algorithm maps efficiently to a 2-D processor array. Systolic architectures provide many advantages for implementation of these algorithms and especially for low level operators which are very common in image processing applications [135].

A key point for the parallel implementation of a chosen algorithm is to verify the characteristics of the image operators used to predict their computational requirements and then to take advantage of their attributes during the implementation.

Low-level image processing operators can be classified as *point operators*, *window operators* and *global operators*, with respect to the way in which the output pixels are determined [136].

Point operations are a class of transformation operations where each output pixel's value depends only upon the value of the corresponding input pixel.

In *global operators* the value of a pixel from the output image depends on all pixels from the input image.

Window operators compute the value of a pixel on the output image as an operation on the pixels of a neighborhood around a corresponding pixel from the input image, using a window mask or kernel and a mathematical function produces an output result [137]. In Figure 3.9 output result for these operators can be observed.

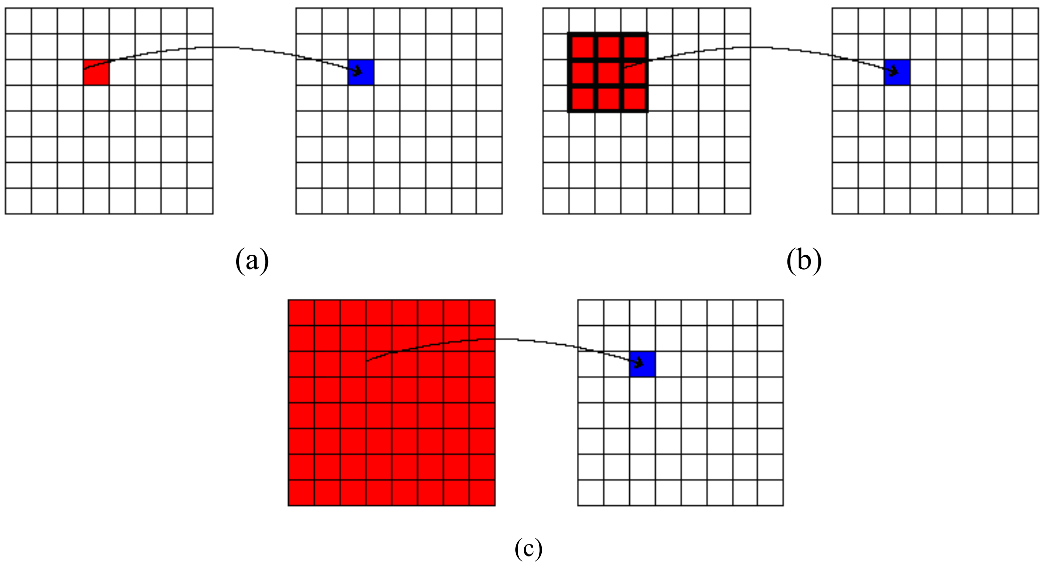


Figure 3.9 Low-level image processing operators: (a) point operators, (b) window operators, (c) global operators

To produce an output pixel, generally a scalar function followed by a reduction function must be applied to the input pixels. The local reduction function reduces the window of intermediate results, computed by the scalar function, to a single result. Common scalar functions include relational operations, arithmetic operations, logical operations and possibly look-up tables. Some common local reduction functions used are accumulation, maximum and absolute value.

The role of this kind of operators is to pre-process images for transformation into symbolic data for high level vision. The values used in the window mask depend on the specific type of features to be detected or recognized. The window mask is the same size as the image neighborhood and their values are constant through the entire image processing. Usually a single output data is produced by each window operation and it is stored in the corresponding central position of the window. The concept of the window operation in image processing is shown in Figure 3.10.

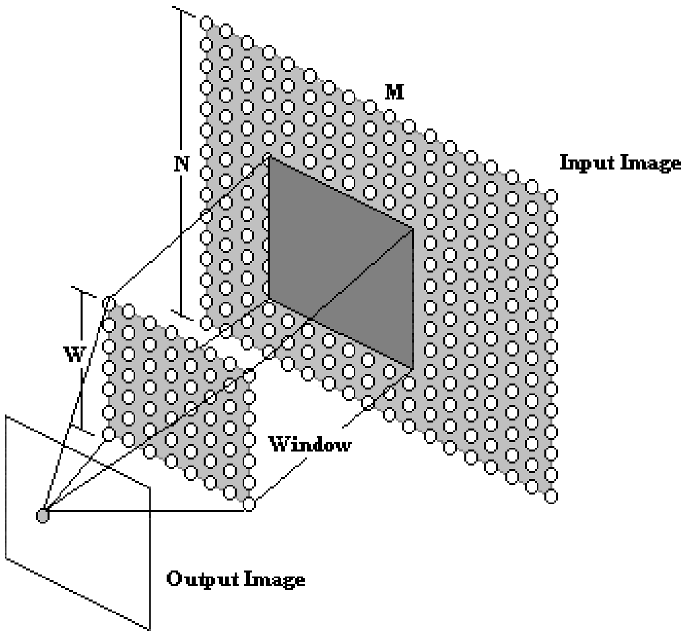


Figure 3.10 Window operation

A certain W × W window operation is performed over the same size area in the input image data. Those identical window operations are repeated over the whole M × N input image data. The different window area is obtained by shifting one column rightward or leftward and one row upperward or lowerward from a window area [138]. For example as shown in Figure 3.11 for two consecutive windows centered at pixel P_{56} and pixel P_{57} respectively.

Several image processing applications perform window-based operators as it is the case of template matching [139], filtering, block matching, 2D feature detection, gray-level image morphology and stereo disparity, motion vector search, among others.

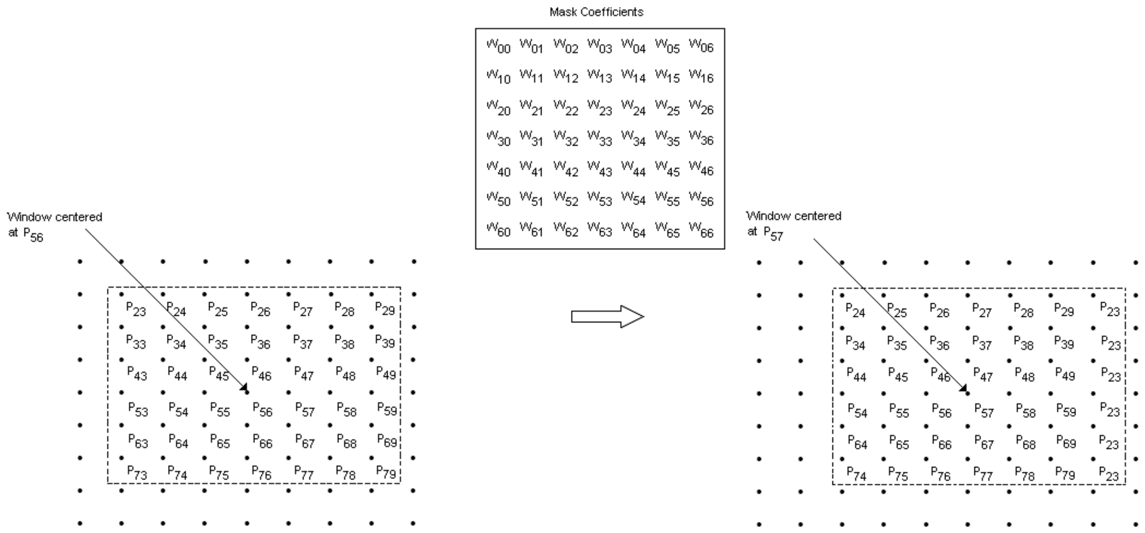


Figure 3.11 Two consecutive output pixels for convolution

3.4.1 Convolution Characteristics

A special case of window-base operators is 2D convolution, which is the basis of many image processing applications [140]. The basic idea of this operator is that a window of some finite size and shape is scanned across the image. The output pixel value is the weighted sum of the input pixels within the window where the weights are the values of the operation assigned to every pixel of the window itself. The window with its weights is called the *convolution kernel*. This leads directly to the following equation:

$$b(x, y) = \frac{1}{s}\sum_{i=0}^{W-1}\sum_{j=0}^{W-1} w(i, j)h(i-x, j-y) \qquad (3.1)$$

Where h is the input image, w is the $w \times w$ mask, s is a scaling factor, and b is the convolved output image. This is computationally expensive. Figure 3.12 depicts the scanning process. In this case, a kernel with a width of 7 is represented with thick black lines.

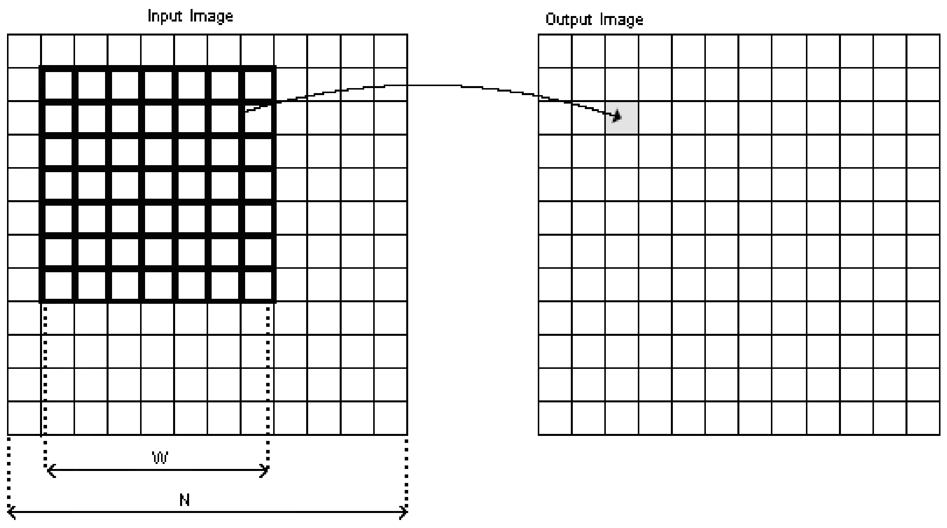


Figure 3.12 Image convolution process

For an M$\times$N input with a W$\times$W convolution mask the total number of operations is M^2N^2 multiplications, $(M^2-1)N^2$ additions, and $(2M^2N^2 + N^2)$ loads/stores [141]. For convolving a 640$\times$480 image with a 7$\times$7 mask, for example, over 50 million operations are required. Thus, for a large-sized mask, the generalized convolution becomes even more computationally expensive because of the quadratic growth of M^2 in the amount of computations. The computational load for an image convolution expressed in terms of arithmetic operations considering a W$\times$W mask on an M$\times$N image is shown in table 3.1.

Table 3.1 Computational load in convolution

Elemental Operations	**Number of Executions**
Multiplication	$W^2 \times M \times N$
Addition	$(W^2-1) \times M \times N$
Load/Store	$(2 \times W^2 + 1) \times M \times N$

According to Equation 3.1, two kinds of operations are required to compute a convolution. A *scalar function* operating on a pixel $h(i\text{-}x, j\text{-}y)$ and a window coefficient $w(i, j)$ to produce a partial result of convolution; and a *local reduction function* to compute a single value from the partial results computed over the window domain, i.e. the number of pixels in the image window. In Equation 3.1, the multiplication represents the scalar function, and the accumulative addition represents the local reduction function [141].

Other image operators in image processing can be described in a similar way as the convolution and they can be generalized as window-based operators. According to the desired output result a different scalar function and local function must be applied as shown in Table 3.2.

Table 3.2 Scalar and local functions common in image applications

Image Application	Scalar Function	Local Reduction Function
Spatial filtering	Multiplication	Accumulation
Convolution	Multiplication	Accumulation
2-D Feature detection	Multiplication	Accumulation
	Absolute difference	
Template matching	Absolute difference	Accumulation
Image morphology	Addition	Maximum
	Subtraction	Minimum
Motion estimation	Absolute difference	Accumulation
Stereo disparity	Absolute difference	Accumulation

Since convolution operations are employed in the early processing stage of many applications and many other image operators can be described in a similar way, the support for fast convolution is essential [142].

Window-based image processing involves high data transfer rates and computational load. This requires an efficient use of communication channel bandwidth and the use of parallel processing to achieve high processing efficiencies. Window-based image processing algorithms require the input image memory to be accessed several times;

therefore the memory access time overhead is a critical parameter design to be considered. To reduce the memory access time, pixels read must be reused in several windows that could be processed in parallel. This is accomplished using pipeline techniques [129, 143] to spread code iterations spatially. Although window-based image operations may access memory in different patterns, they share important features that can be generalized [144]:

- Image operations are memory intensive and at least one new pixel of data is typically needed for each step in the computation.
- A high potential for parallelism is available since window operations include a large percentage of independent operations that are applied to each pixel.

Image data is normally represented in linear memory organizations, in this way neighboring pixels in the image are not necessarily stored as neighboring elements in memories. This fact obscures the spatial relationship between the pixels so the 2D data parallelism is not explicit. If the spatial data dependencies are exposed, it should be easier to implement a hardware architecture to use uniform two dimensional data access patterns that are needed for data parallel execution. A related issue is that data for window operators usually overlap with the neighbor windows of the surrounding pixels. This means that there is a great deal to create simple vectors of data elements that can be processed by parallel vectorization techniques.

3.5 Systolic Arrays in FPGAs

By analyzing several representative problems in computer vision, the following architectural requirements are observed:

- *Computational characteristics.* For low-level algorithms, SIMD or Systolic and fine grained architectures are implemented.
- *Communication*. At a lower level, communication is limited to a local neighborhood
- *High bandwidth I/O*. A typical image contains a large amount of data; therefore a high bandwidth I/O is essential to sustain good performance.
- *Resource allocation*. Speeding up only one stage of a vision system will not result in a significant speedup of overall performance. Hence, appropriate computational resource should be allocated to all the stages of the algorithm.
- *Load balancing and task scheduling*. For good performance, the load on different processors should be balanced.
- *Fault tolerance*. In a multi-processor system, failure of some processing elements should not result in an overall system failure. Therefore, a graceful degradation should be supported.
- *Topology and data size independent mapping*. Often, a specific processor topology is preferred for an algorithm depending on its communication characteristics. Consequently, flexible communication support is essential for mapping many communication patterns. The algorithm mapping should be independent of data size.

Regarding the characteristics mentioned above, the present research work is based on the design of a systolic array which is appropriate to implement low level image processing algorithms. Systolic arrays are used quite frequently in the domain of massively parallel computing, since they readily map to parallel problems while also limiting communication bottlenecks found in other parallel processor architectures [145].

Designing a systolic array using reconfigurable devices, involves designing the internal workings of each individual processing element in addition to designing the dataflow and communication link between processors in the array. The systolic elements or cells are customized for intensive local communications and decentralized parallelism. Because an array consists of PEs of only one or, at most, a few kinds, it has regular and simple characteristics. The array usually is extensible with minimal difficulty. Data pipelining reduces I/O requirements by allowing adjacent PEs to reuse the input data.

In order to implement a systolic array using an FPGA some performance tradeoff must be taken into account because these determine the array's performance efficiency. Among other it can be mentioned the level of granularity and the extensibility.

The level of PE granularity directly affects the array's throughput and flexibility and determines the set of algorithms that it can efficiently execute. Each PE's basic operation can range from a logical or bitwise operation to a word-level multiplication or addition to a complete program. Granularity is subject to technology capabilities and limitations as well as design goals. Because systolic arrays are built of cellular building blocks, the PE design should be sufficiently flexible for its use in a wide variety of topologies implemented in a wide variety of substrate technologies.

In the particular case of a window-operation some extra characteristics must be taken into account in order to carry out the FPGA implementation. The first issue is the number of processing elements to be used in the array, when the algorithm to be implemented is chosen, it needs to be tuned to the special hardware being used; therefore the number of PEs used in the systolic array needs to be selected. The second issue is the size of the storage elements which depends on the number of rows to be processed in the image and the amount of parallelism. The third issue is the complexity of PEs. Operations implied in window-operators require multipliers; therefore the word length, the cycles per operation as well as the number and kind of operations must be selected to assure reduced area occupation.

The flexibility enabled by the FPGA technology is used in the proposed work to explore the architecture space in the context of varying the aforementioned system parameters. In order to achieve high throughput, a good tradeoff between flexibility, performance and area occupancy under real time conditions must be achieved which implies to tune these parameters to find the optimum operation. In this thesis some graphs of behavior are obtained to determine the point in which the speed is the highest and the area occupied is the lowest.

The coefficients used to determine the operator that executes the architecture in a given moment can be defined by the user. This customization drives to the implementation of different algorithms leading to an inexpensive “programmable” systolic array hardware architecture, which constitutes one of the main advantages of the proposed schema.

Additionally to this advantage, the FPGA technology presents other benefits to implement systolic arrays. FPGA provides the means to create state-of-the-art System on Chip (SoC) designs in a fraction of the time previously required; it is possible to create migratable designs from one silicon fabrication process to another. In summary the proposed architecture present the following advantages:

- The same physically implemented systolic array can execute different types of algorithms, which enables versatility of the physical implementation.
- Algorithms can be changed in real-time, leading to greater flexibility in the environments that depend on multiple algorithms to solve the problem.
- The architecture is scalable and parameterizable so that it can be easily used for new complex applications
- The hardware implementation is suitable for SoC integration in embedded systems, which is especially true for applications that depend on numerous filtering and image processing algorithms and can thus be executed on a single programmable processor array.

This thesis uses FPGAs in the systolic approach because they offer a design route which is both fast and manageable and at the same time they provide performance near some ASICs and the flexibility of software-programmable substrates [146, 147].

Chapter 4

Architecture Overview

In this chapter an overview of the flexible systolic-based architecture proposed is presented, where the key feature is the implementation of a generic array of processors as it provides enough flexibility to run a variety of low-level processing algorithms and constitutes a platform to pursue the implementation of higher complexity algorithms.

The hardware implementation includes the proposal of some enhancements to overcome the lack of flexibility inherent to systolic arrays, in the sense that it is impossible to produce an array to match all the possible sizes of different problems, and to extend the architecture performance to generalize the systolic array functionality. Such enhancements include a mechanism to reduce the number of access to a global memory using a new schema based on local storage buffers and Router elements to handle data movement among different structures inside the same architecture. These two components interact to provide the capability of process chaining. Input image buffers can be accessed by different processing blocks while output image buffers containing partial results can be accessed by cascading processing blocks to reuse data. The global buses implemented reduce routing problems and constitute a means for parameters interchange inside the architecture to customize the number of processing elements used as well as the kind of operation performed by the processing elements for a particular application.

Using FPGAs allows the computational capacity of a hardware implementation to be highly customized to the instantaneous needs of an application. In the architecture presented here, the number of PEs in the systolic array can be modified as well as the memory size for buffers used in a particular problem, keeping low memory capacity per PE. The architecture can manage fixed-point operations achieving enough precision for many applications and can be extended to carry out floating-point operations.

4.1 Functional Requirements

Convolution is a fundamental mathematical operation to many common image processing operators. In a convolution operation, two arrays of numbers with different sizes are multiplied together to produce a third array of numbers. In image processing, convolution is used to implement operators whose output pixel values are linear combination of certain input pixels of the image. Convolution belongs to a class of algorithms called window operators which use a wide variety of masks, to calculate different results, depending on the desired function.

The basic idea of window operations is to use a window of fixed size to scan over an image. The output pixel value is the weighted sum of the input pixels within the window where the weights are the values of the operation assigned to every pixel of the window. The window with its weights is called the convolution mask. Mathematically, convolution on image can be represented by the Equation 3.1.

Based on the convolution operator some functional requirements for the architecture implementation are stated, and then they are generalized to support a set of the common image processing application based on window operators presented in Table 3.2.

According to Equation 3.1 the convolution is the sum of products of each pixel and corresponding mask coefficient. It can be observed that main operations to be implemented are multiplication and accumulation according to Table 3.2; PE's structure is based on this requirement. From Equation 3.1 it can also be observed that convolution presents a high potential for parallelism due to the fact that the arithmetic operations involved can be executed independently to each pixel or image region. Convolution is carried out in a two dimensional space which makes it suitable to be implemented using a parallel systolic array architecture, which achieves a good tradeoff between parallelism, regularity and execution time. In this work an array of size $W \times W$ is adopted, where W is the size of the moving window for a kernel with a width of 7. Using this architecture, multiplication and accumulation operations can be completed in one clock cycle aiming for high throughput. Although multiplication usually requires large hardware resources and long execution time, dedicated multipliers in modern FPGAs facilitate the use of one fast multiplier processing element.

As mentioned previously, any 2-D convolution operation involves passing a 2-D window mask over an image, and carrying out a calculation at each window position as observed in Figure 4.1. Taking this into consideration four parameters must be specified to complete the implementation. The size of the window, the mask coefficients depending on the operation to be performed, the scalar function used during operation and the local reduction function to be applied to each extracted window of the input image to produce an output result. These parameters determine PEs functionality design.

From Figure 4.1 it can be deduced that when an output pixel is computed, access to entire previous rows or portions of previously input rows of input pixels are needed. In this way image memory must be accessed several times in order to complete a computation. For processing purposes, the straightforward approach is to store the entire input image into a frame buffer, accessing the neighborhood's pixels and applying the function as needed to produce the output image. If real-time processing of the input images is required for a $W \times W$ window mask, $W \times W_{.}$ pixel values are needed to perform the calculations each time the window is moved and each pixel in the image is read up to $W \times W_{.}$ times. The memory bandwidth constraints make impossible to obtain all these pixels stored in the memory in only one clock cycle, unless any kind of local caching is performed.

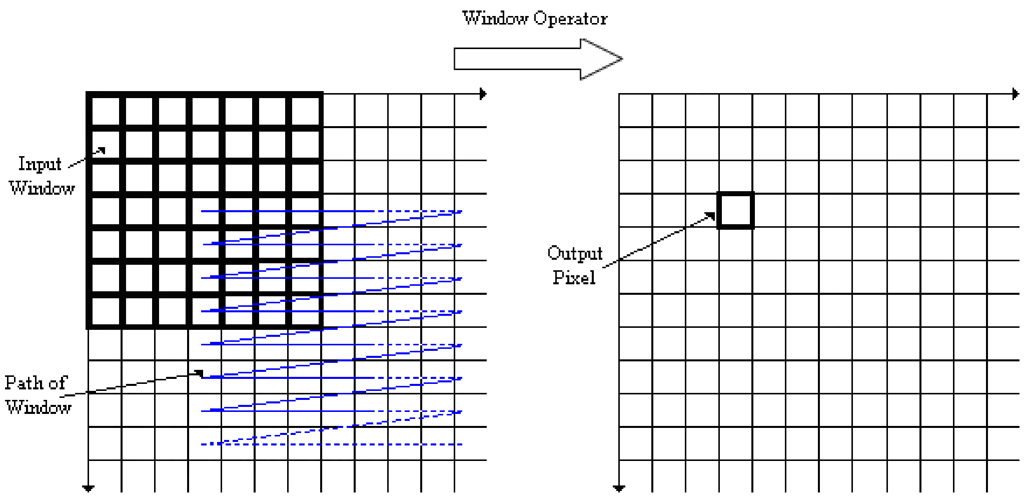


Figure 4.1 Image scanning process for window operators

Traditional approaches are characterized by their abundant memory directly connected to each processing element. In this thesis a circular buffers schema is used to manage the traffic of data to and from the processing array. Using memory address pointers it is possible to keep track of the elements being processed.

Input data from the previous W rows can be cached using the memory buffers till the moment when the window is scanned along subsequent rows. To reuse hardware resources when new image data is processed, a shift mechanism between buffer rows is used. Data inside the buffer can be accessed in parallel, reducing the time needed for data reading, this schema assures that each input image pixel is fed only once to the FPGA. Buffer elements synchronize the supply of input pixel values to the processing elements, moreover image buffers allow performing several window operators in parallel and they add the possibility to carry out computations with local data. Instead of sliding the window across the image, this implementation feeds the image through the window as can be observed in Figure 4.2. The complete process will be presented in detail in the following sections.

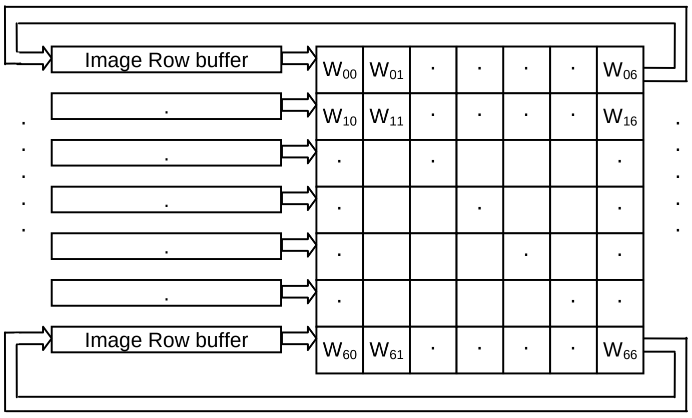


Figure 4.2 Structure for image buffers.

Introducing the row buffer data structure adds additional considerations. With the use of both caching and pipelining there needs to be a mechanism for controlling data flow through buffer rows to prime, place, and take out image synchronously. Another important issue to be considered is a good tradeoff between resource allocation and level of parallelism, the number of window masks processed in parallel, depends on the number of input image rows stored.

Input images are stored as a collection of discrete pixels arranged in a two dimensional space, each value has an associated row and column to indicate its coordinates (position) where each pixel's value represents gray levels for that coordinate. The gray levels are usually represented with a byte or 8-bit unsigned binary number, ranging from 0 to 255 in decimal. These values determine the I/O buses size.

Internal buses and internal registers size are determined by the operators' word length and operation applied. For some window operation floating-point operations are required and word length is a function of the precision used.

The format of the floating-point numbers depends on the application requirements and expected bounds. Providing excessive resolution comes at a high price and should not be spent if not needed. Truncation and rounding effects must be considered when dealing with either fixed-point or floating-point formats. These effects introduce an error value which depends on the word size of the original value and how much of the word is truncated or rounded. Floating-point operations are rather difficult to implement using FPGAs, they require big amounts of area and they are inherently slow because of their complexity. The fixed-point arithmetic operations are relatively faster compared to the floating-point operations, this number representation facilitates implementation of most of the calculations as integer arithmetic, as little pre or post-normalization is required. In this thesis gray level input images are used with 8-bit unsigned binary number format, mask coefficients uses a 9-bit signed format and computations are performed using fixed-point with an 8.8 representation with a resolution of 0.00390625 according to the analysis that will be presented in section 5.2.

Previously, there has been several hardware architectures reported in the literature to implement window-based image processing algorithms [10, 11, 141]. However, most of them focus on the performance of a single image processing algorithm, usually 2D image convolution, without considering implementation aspects such as flexibility and silicon area. When different window-based algorithms are considered, they are processed as independent functions. Furthermore operations which need to be executed on the outputs of convolution stages cannot be performed on the same hardware at the same time; as a consequence data results are not reused [141].

In order to overcome this shortcoming, in this thesis a processes chaining schema is presented. To reuse results from different processing stages a cache schema is required. Output buffers with similar characteristics to the input image buffers have been implemented to store the results of a particular processing stage.

Processing blocks can be replicated inside the architecture. Internally, each block has a common basic structure and constitutes a customizable component through a set of parameters to select options for various processing features such as window operation, window mask coefficients, cache size, data type selection, and fixed-point processing. A Parameters bus provides the facilities to send option values to the FPGA to be configured. Routers are responsible for controlling the flow of data into and out of the processing blocks. Router simplifies processing block design and facilitates design reuse.

Using the propose schema it is possible to chain processes since different processing blocks inside the same FPGA can carry out a different window-operator over the same data set.

4.2 General Overview of the Architecture

In this section the proposed hardware architecture for window-based image processing is presented. This architecture is based on a systolic array that makes use of extensive concurrency, data reuse, and reduced data bandwidth. The design is aimed to present a modular and scalable solution.

The design approach has two major steps, definition of a model of the architecture in terms of a few parameters i.e. abstract model, and accomplishment of the final architecture by a tradeoff analysis driven by the simulation process. The architecture is defined by the following parameters:

- Number of PEs,
- Interconnection of the PEs
- Bandwidth between processors array and main memory
- Bandwidth between processors array and host PC
- Local memory in each processing block
- Window mask size

A simplified block diagram of the proposed hardware architecture is shown in figure 4.3. The architecture is composed of six main blocks:

- A high level control unit
- An external main memory
- A dedicated processor array
- Routers
- Image buffers
- Internal buses

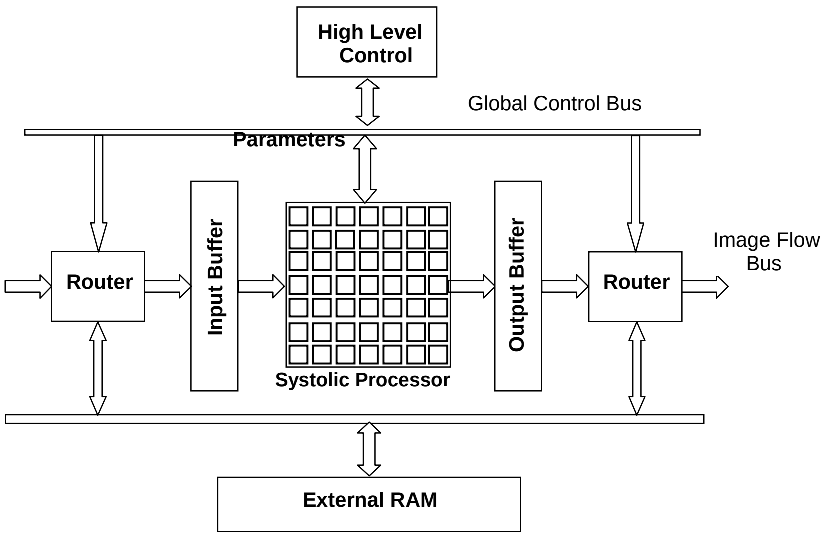


Figure 4.3 Block diagram of the architecture.

High Level Control Unit: This unit is placed in a host PC. The main purpose of the control unit is to manage the data flow and synchronize the different operations performed in the architecture. The high level controller starts and stops the operation in the system, furthermore, it is responsible of image capturing and displaying. From the PC it is possible to choose a particular operation that can be performed by the PEs in the systolic array, to coordinate operations and to manage bidirectional data flows between the architecture and the PC. From this unit, the user can select configuration parameters to customize the architecture functionality; the parameters include the size of the images to be processed, the coefficients for the mask to be used during processing and the kind of arithmetic to be employed between integers or fixed-point.

Main Memory: The memory in the architecture is a standard RAM memory for storing data involved in the computations. The data in the memory are accessed by supplying a memory address. The use of these addresses limits the bandwidth to access the data in the memory, and constrains the data to be accessed through only one memory port. Furthermore, the time to access the data is relatively long, therefore a buffer memory is included to store the data accessed from memory and to feed the processor array at a much higher rate. The buffers are used to re-circulate the data back to the processors, and they reduce the demand on main memory. An important issue to be solved is the allocation of area to implement data buffers. To obtain good performance one of the issues in the architecture design is, therefore, how to schedule the computations such that the total amount of data accesses to main memory is bounded.

Processor Array: The processor array is the core of the architecture where the PEs are organized in a 2-D systolic approach; and where the algorithms are executed. The processor array obtains image pixels from the buffers, and mask’s coefficients from memory to start a computation. The processing array achieves a high performance due to a pipelined processing schema and local connections without long signal delays. The array organization with a small number of boundary (I/O) processors reduces the bandwidth between the array and the external memory units. The control unit specifies and synchronizes the actions to be performed in the PEs.

Routers: The Router is responsible for all data movement in and out of the systolic array as well of interfacing processing modules to external memories. The data streams routers take data from/to input/output image memories and make explicit the data parallelism usually found in the image processing. The incoming data is stored in external memory RAMs and data is brought into a set of internal buffers prior to be processed in parallel. The produced data by a processing block can be stored and then transmitted to an external memory output using a router.

Buffers: The purpose of the buffers is to supply data to the processor array and mask the long main memory latencies. The buffers have a fixed amount of storage to keep some rows of the input image or the intermediate data from a processing module. The storage buffers are organized in a First Input, First Output (FIFO) style. In each clock cycle, the data present at the buffers are sent to the processors array or to the main memory. Address decoding for the buffer is executed using pointers that make reference to the buffer's row that is being processed or being filled. These pointers allow a circular pattern in data movement inside the buffers. The buffer basically carries out the following operations:

- Prefetch data from the main memory into its rows to hide the memory latency
- Reorders the information according to the processing needs of the algorithm to increase parallelism
- Stores intermediate information for its reutilization in subsequent processing blocks

Internal Buses: The global bus interconnects architecture elements to interchange back and forward control or configuration information, i.e. mask coefficients. In addition, this bus is connected to the high level control unit placed in a Host processor which is in charge of data and parameters transfer via Direct Memory Access (DMA) with the processor.

This architecture schema resembles a high level pipeline schema, formed of memory units and computing units. The architecture is intended for data communication among processes using data buffers abstraction. With these elements it is possible to chain processes since different processing blocks inside the same FPGA can carry out a different window-operator over the same data set. The results obtained by each block can be stored in the output image buffers and reused by subsequent processing blocks. This structure of cascading interconnection is a key feature of the architecture since it supplies generality to the array of processors, providing enough flexibility to run a variety of low-level processing algorithms and constitutes a platform to pursue the implementation of higher complexity algorithms.

4.3 Functional Description of the Architecture

Due to image processing algorithms map efficiently to a 2D processors array, therefore the proposed architecture consists of a main module based on a 2D, customizable systolic array of $W \times W$ PEs as shown in the block diagram in Figure 4.3.

Input image pixels are read from an external memory bank where they are stored in a linear organization together with the mask coefficients, then they are placed in an internal repository implemented using double port BlockRAM memories as neighboring elements. Input buffers are operating in a circular pipeline and they are able to parallelize data to be used by the systolic array which can compute, in parallel, several window operations.

To manage the data flow among the blocks inside the architecture, a Router element has been provided. The router module allows data transfer directly to a processing block or to a storage element into the FPGA before being processed. The Router module provides a flexible and scalable solution for routing data between buses, systolic array and local row memory.

The global bus allows interconnection between architecture elements to interchange back and forward control or configuration information, i.e. mask coefficients. In addition this bus is connected to the high level control unit placed in a Host PC which is in charge of data and parameters transfer via DMA with the main memory.

The Host PC starts and stops the operation in the system, furthermore, it is responsible of image capturing and displaying. From the PC it is possible to choose a particular operation that can be performed by all the PEs in the systolic array.

The architecture operation starts when a column of pixels from the input image is broadcasted to all the PEs in the array. Every PE working in parallel keeps track of a particular window operation. At each clock cycle, a PE receives a different window coefficient, stored in an internal register, and an image pixel coming from the input image buffer. These values are used by every PE to carry out a computation, specified by a scalar function, and to produce a partial result of the window operation. The partial results are incrementally sent to the local reduction function implemented in the PE to produce a single result when all the pixels of the window are processed. PE's supports most window-based operators in image processing: multiplication, addition, subtraction, accumulation, maximum, minimum and absolute value.

The produced result is stored in the output image buffer to be reused for a subsequent processing block chain connected or it can be sent to an external memory. Once a result is produced by a processing element, the PE is ready to start a new computation. The processing elements work progressively in the same way until the entire image has been scanned in the horizontal and vertical direction. The processing elements are continuously being reused for computing different windows along the input image.

For simplicity the control unit for the systolic array has not been show in Figure 4.3. This module is in charge of generating all the control and synchronization signals for the elements of the architecture. The unit synchronizes external memory, input and output buffers bank, and systolic array computations. The control unit indicates which processors execute an operation and when a result must be sent to the output storage elements.

4.4 Data Movements and Memory Schema

The hardware architecture was designed with the following characteristics to allow efficient image processing:

- Provides enough abstraction to the user of the hardware about the details inside the architecture
- Supports pipelined processing.
- Allows multiple processing blocks to be implemented simultaneously.
- Presents an efficient and simple memory model.
- Combines an efficient memory structure with a flexible interconnection mechanism to offer a scalable solution

Among the key elements used in the architecture, the global image bus facilitates the communication for multiple processing blocks simultaneously inside the same FPGA. The control bus architecture consists of a shared global bus that interconnects all the elements inside the architecture to interchange back and forward control or configuration information, i.e. mask coefficients. The control bus executes basically tree main functions:

- **Parameter Access** - Any run-time information required by the architecture can be transferred using the global bus. Used in this way, the global bus implements the parameter path.
- **Control of the Routing** - The global bus is used to instruct each Router element where to route the image data i.e. flexible pipelined routing.
- **Configuration of the PEs** - Some parameters can be used to configure the PEs according to the desired application.

Besides the buses another important element for the movement of the information inside the architecture is the Router. Router elements in the architecture constitute the intra FPGA communication system. They have configurable routing capabilities for communication between systolic array units and their memory, other systolic array units with each other, and control structures.

The Router provides a flexible, scalable solution for data routing. The Router is responsible for three tasks:

- Accessing the architecture buses.
- Generating the input image data flow for the processing blocks.
- Handling the output image data flow for the processing blocks.

The Router has also the capability to present the image data in the format in which the architecture expects it.

Due to the structure of the convolution algorithm, for each successive output pixel, it is required the access to the entire previous rows or portions of previously input rows of input pixels. In this way image memory must be accessed several times in order to complete a computation.

The system has been provided with a simple memory model that reduces the number of accesses to external memory and allows the access in parallel to the input image data.

A small image buffer implemented with two port BlockRAM memories has been used to keep some rows of the input image. This buffer is operating in a circular pipeline, where image pixels are stored as neighboring elements.

Routers take data from the input image memories and transfer data to the input buffers that store as many rows as the number of rows in the mask used for processing a window. An additional row is added to the buffer to be filled with new image data in parallel with the rows being processed; in this way the memory access time is hidden. Each time a window is slid in the vertical direction, a new row in the buffer is chosen to be refreshed with input image data, following a FIFO style. When the buffer end is reached, the first buffer row is reused following in this way the circular pattern as shown in Figure 4.4.

The coefficients of the window mask are stored inside the architecture in a memory bank that is able to shift data from one element to its neighbor. A shift register bank is distributed on internal registers of the processing elements to delay the mask coefficients.

In a similar way to the one used to read the input data, the memory containing the coefficients of the window mask of a window operator is read in a column-based scan. Figure 4.5 shows the reading process of the mask coefficients as time progresses. The coefficients are read sequentially and their values are transmitted to different window processors when an image is being processed.

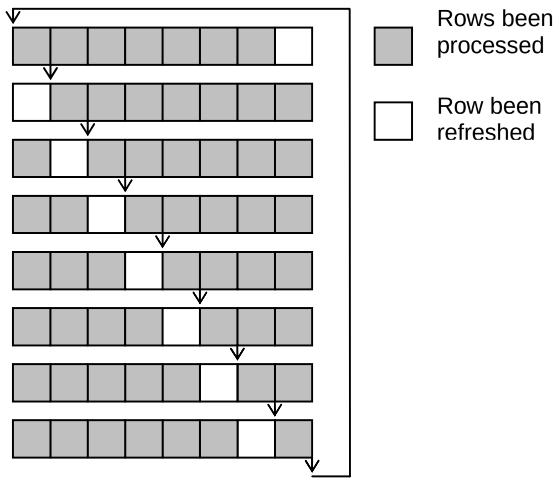


Figure 4.4 Circular pipeline in the buffer memory

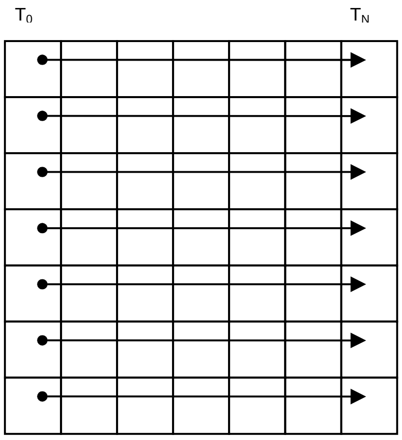


Figure 4.5 Reading pattern for window mask

The reading process of the window mask coefficients and input image pixels requires a synchronization mechanism to match the operations sequence.

4.5 Detailed Description of the Architecture

The main process block in the architecture is conformed by a 2D systolic array. Figure 4.6 shows the way in which PEs are interconnected for a window mask of 7×7 elements.

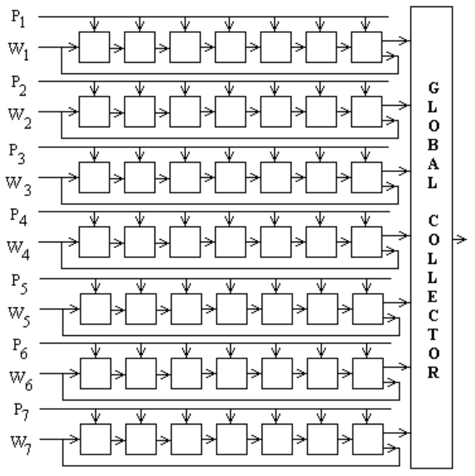


Figure 4.6 2D systolic array implementation

Every block in Figure 4.6 represents a processing element. PEs are interconnected in a two dimension array and they are organized to process several windows in parallel in the vertical direction. The processing element's structure has been defined from the convolution operation definition.

4.5.1 Processing Element's Structure

As it has already been stated in chapter 3, convolution is outlined as:

$$b(x, y) = \frac{1}{s} \sum_{i=0}^{W-1} \sum_{j=0}^{W-1} w(i, j) h(i - x, j - y) \tag{4.1}$$

According to Equation 4.1, two kinds of operations are required to compute a convolution. A *scalar function* operating on a pixel $h(i\text{-}x,\ j\text{-}y)$ and a window coefficient $w(i,\ j)$ to produce a partial result of convolution. A *local reduction function* to compute a single value from the partial results computed over the number of pixels in the image window. In this equation, the multiplication represents the scalar function, and the accumulative addition represents the local reduction function.

Each PE requires two operational inputs, one pixel from the input image and a coefficient from the window mask. PE internal structure comprises an ALU element in charge of performing multiplication and one accumulator register in charge of implementing accumulative addition. If the value of convolution is desired at the point (x, y), the center of the mask is placed at (x, y). A multiplication of the image pixels and mask values is computed. This operation is followed by an accumulative addition reduction operation. Another register is required in order to perform the accumulation of these intermediate results. In the architecture a partial results collector (PRC) has been implemented for this purpose in each PE column. The basic set of these operations is repeated at all possible (x, y) location inside the whole window as can be observed in Figure 4.7 for an image of finite size, $\mathrm{M}\times\mathrm{N}$ and a mask of finite size $W\times W$. When the final result is obtained, it is stored in a global collector register before being sent to the output memory.

In order to provide support to other window-based operators presented in most common image processing algorithms, the ALU functionality has been extended to implement all the basic operations described in Table 3.2. A control word inside each PE selects the appropriate operation to be performed.

$(x\text{-}w, y\text{-}w)$	…	$(x+w, y\text{-}w)$
. . .	(x, y)	. . .
$(x\text{-}w, y+w)$	…	$(x+w, y+w)$

Figure 4.7 Convolution operation

The structure for each PE that provides support to the operations involved in most window-based operators in image processing is presented in Figure 4.8.

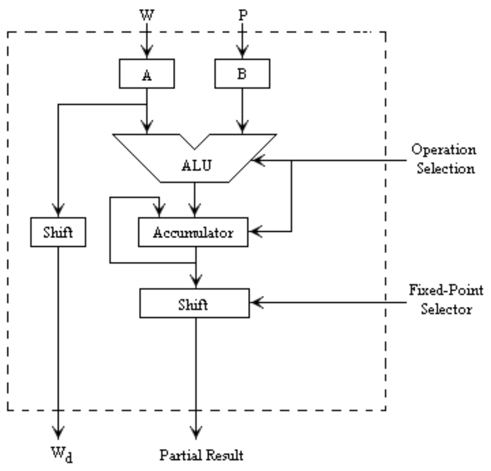


Figure 4.8 Processing element implementation

One PE comprises an arithmetic processor (ALU) that implements the scalar functions and a local reduction module (Accumulator). PE can be configured by a control word selected by the user according to a desired algorithm; the ALU provides the functions of multiplication, addition, subtraction, and absolute value. The Accumulator module implements the local reduction functions of accumulation, maximum, minimum and absolute value. Each PE has two operational inputs, pixels from the input image and coefficients from the window mask denoted by P and W, respectively in figure 4.8. Each PE has two output signals, the partial result of the window operation and a delayed value of a window coefficient that is transmitted to its neighbor PE. Every clock cycle, each PE executes three different operations in parallel [144]:

- Computes the pixel-by pixel value to be passed in the next computation cycle.
- Integrates the contents of the outputs register calculated at the previous clock cycle, with the new value produced in the arithmetic processor (ALU).
- Reads a new mask coefficient and stores it into the register. Then, PE transmits the previous coefficient to the next PE.

In equation 4.1 *s* represents a scaling factor. As has been already mentioned, it is required in several window-based algorithms. Operations with this factor introduce floating-point values. For example in the particular case of Gaussian filtering, the mask coefficients are represented using decimal numbers produced from a vector of weights that approximate a discrete Gauss function.

The use of floating-point mathematics is often the most important issue to solve when creating custom hardware to accelerate a software application. Many software applications make liberal use of the high performance floating-point capabilities of modern CPUs, whether the core algorithms require it or not. Although floating-point arithmetic operations can be implemented in FPGA hardware, they tend to require a lot of resources. Generally, when facing floating-point acceleration, it is best to either leave those operations in the CPU portion of the design or change those operations to fixed-point.

A detailed description of the process for converting from floating-point to fixed-point operations can be highly algorithm-dependent, but in summary, the process begins by analyzing the dynamic range of the data going into the algorithm and determining the minimum bit-width possible to express that range in an integer form. Given the width of the input data, it can be traced through the operations involved to determine the bit growth of the data. For example, to sum the squares of two 8-bit numbers, the minimum width required to express the result without loss of information is 17 bits. The square of each input requires 16 bits and the sum contributes 1 more bit. By knowing the desired precision of the output, working backwards through the operation of the algorithm it is possible to deduce the internal bit widths. Well-designed fixed-point algorithms can be implemented in FPGAs quite efficiently because the width of internal data paths can be tailored. Once the width of the inputs is known, internal data paths, and outputs, the conversion of data to fixed-point is straightforward, leading to efficient implementation on both the hardware and software sides. The width should be large enough to introduce acceptable quantization error according to the constraints of the algorithm.

In order to fulfill the requirements of such algorithms the PEs have been designed to support applications that require operation with fixed-point arithmetic. Fixed-point operations are faster than floating-point calculations but provide less precision.

Addition and subtraction can be done in similar ways as the integer addition and subtraction. However, in order to preserve the accuracy; multiplication and division require larger temporary accumulators with twice the word length. Therefore, to eliminate the growth in the size of internal registers; multiplication can be performed by scaling the operands multiplying in powers of 2, which can be easily accomplished with shift movements. A shift register is included in each PE to scale results when applications require fixed-point operations.

4.5.2 2D Systolic Array

Several PEs interconnected one after another as shown in Figure 4.9 conform a column of the array. In this structure a PRC is included in order to keep partial results as explained in previous section. The number of PEs in the column depends on the parallelism desired for a given application. For a column of seven PEs, seven pixels can be processed in parallel due to data from window mask and the input images are available in one clock cycle.

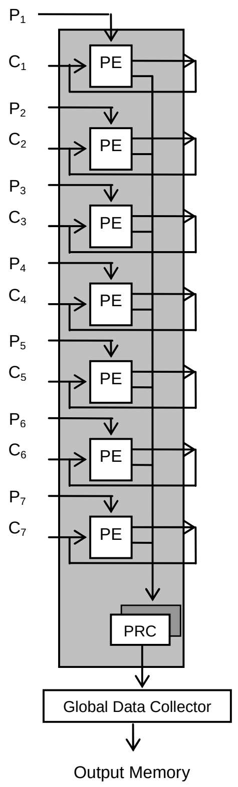


Figure 4.9 Column of processing elements

Partial results in the PRCs are sent out of the column to a global data collector (GDC). The PRC scans progressively the PEs column according to a control signal to deliver the PEs' results progressively.

In order to extend parallelism to 2D some columns of PE's are connected in cascade. Figure 4.10 shows the 2D systolic organization of the columns of PEs for a 7×7 array. This structure allows to process seven processing windows in parallel.

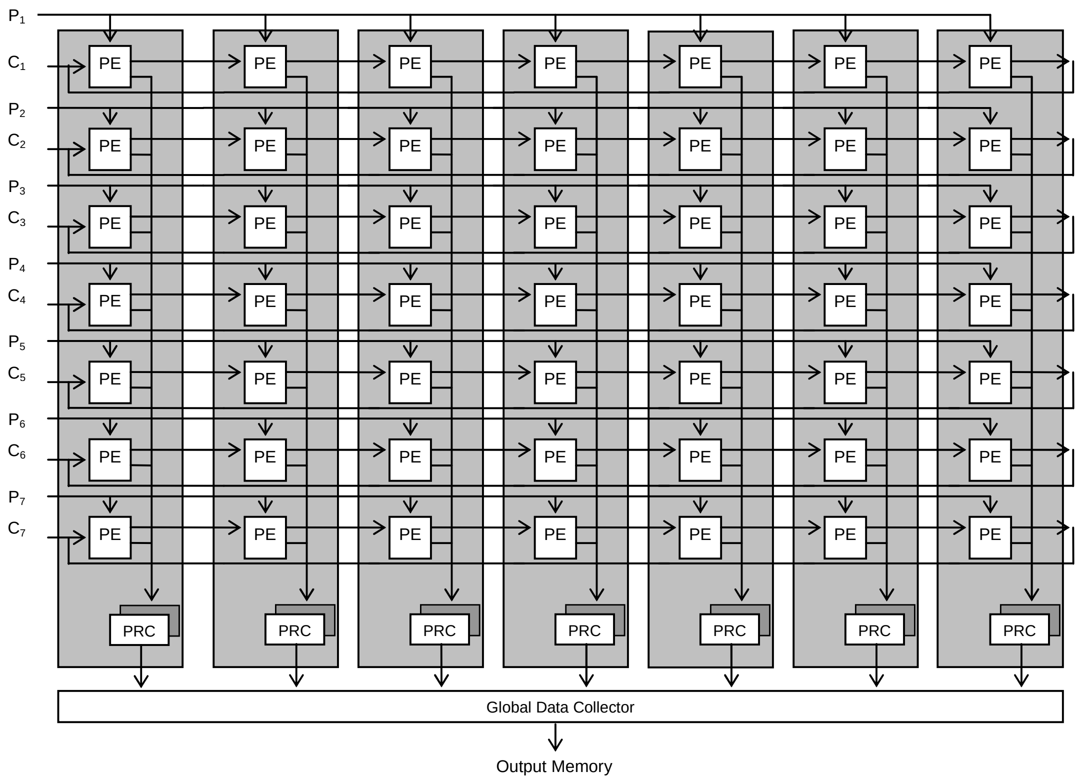


Figure 4.10 Systolic array main modules

All the PEs in a column receive a column of pixels from the input image via the input buffer and a column of mask coefficients that is shifted from left to right between array columns every clock cycle following a circular pipeline schema. The PEs are activated every clock cycle, following also a pipeline schema as shown in Figure 4.11 for a systolic array of 7×7 PEs.

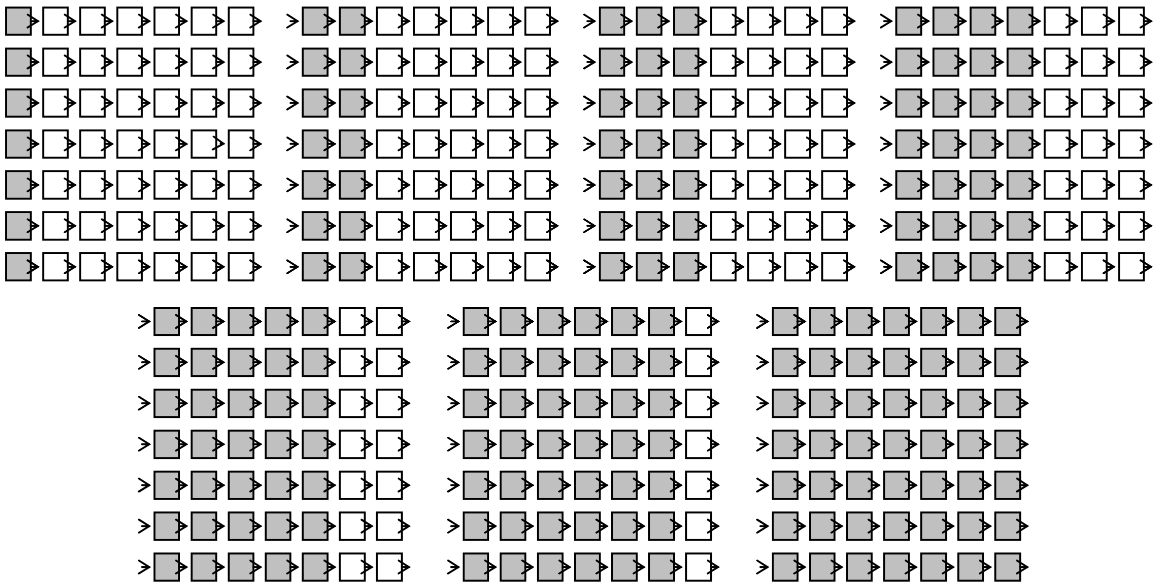


Figure 4.11 PE's activation schema

After a short latency period, all PEs in the array are performing a computation according to a control word. From that moment on, each new column of pixels sent to the array shifts the convolution window to a new adjacent position until the whole image has been visited in the horizontal direction.

Reading image pixels from the buffer one row below, it is possible to cross the image in the vertical direction. The image buffer is updated during PEs operation, in a circular pipeline schema too.

The partial results obtained by each PE's are stored in the partial results collectors and the final result is sent out of the array to a global data collector. The global data collector verifies progressively the PRCs from right to left according to a control signal. After a certain latency period has elapsed, the PRC delivers a result progressively. Then, the result passes to the global data collector in order to be written in the output memory.

Chapter 5

Hardware Implementation

This chapter describes the proposed FPGA-based platform architecture that has been designed specifically to support window-based operators. The architecture is comprised of two parts: a *hardware substrate* and a *software interface*.

The hardware substrate supports the computation engine which constitutes the problem solver of the platform and has been designed to provide real-time performance and enough flexibility to allow process chaining. The software interface allows modifying the function described by the computational part. Two types of adaptation are available: major structural modification and parameter tuning.

The architecture modularity, a structural change, can be modified according to the number of Routers presented in the architecture. Parameter tuning consists in choosing the configuration values to define data flow inside the architecture, operation to be performed by PEs and data flow into and out of the processing elements. The software interface resides in the Host PC and allows the function and parameters selection, the order in which algorithms are going to be performed. The interface is in charge of images capturing and displaying.

Physically the FPGA platform has been implemented using the Celoxica RC1000 development system shown in Figure 5.1. The RC1000 board is a PCI bus plug-in card for PCs. It contains a Xilinx Virtex-2000E FPGA, 8 Mbytes of external SRAM divided into 4 separate independently accessible banks and a 32-bit PCI interface to the host PC. Data can be loaded from the PCI into the SRAM banks and then read as required by the FPGA. DMA, data buffering and clock speed control make it suitable for implementing high-speed image processing applications.

In this chapter a detailed description of the internal structure of the principal modules of the architecture is provided as well as the results from the synthesis process. In addition details of the fixed-point implementation are presented considering a tradeoff between the error and the amount of area occupied. Finally a performance analysis of the architecture is presented considering parameters such as frequency, level of parallelism, and the number of processors to obtain graphs of behaviour that represents the architectural advantage of the proposed schema. In order to estimate the scalability and the level of improvement in performance of the architecture, it has been synthesized to different technologies.

5.1 RC1000 Prototyping Board

Currently, there are a lot of different FPGA prototyping boards that can be found in the market, with each having different architecture and FPGA chips installed. For this thesis implementation, a RC1000 development board specially made by Celoxica for use with the DK4 development suite that includes a Handel-C compiler has been used. The practical advantage, when using this board, is the packaged communication library that is written in Handel-C. The communication library provides pre-built hardware design to gain access to the PCI bus and the on-board memories of the RC1000. The block diagram for the board is shown in Figure 5.1.

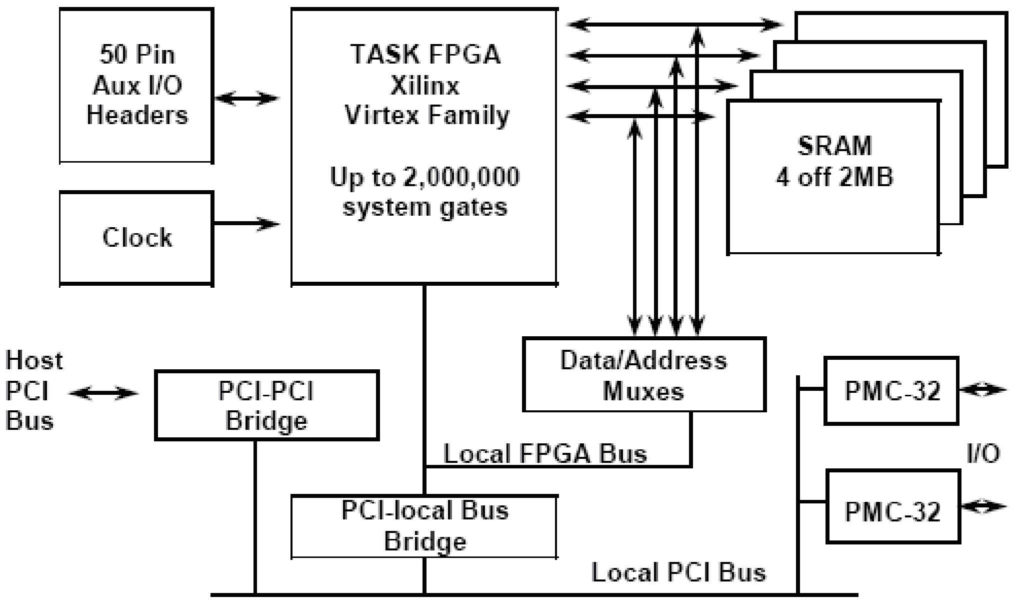


Figure 5.1 RC1000 block diagram

The RC1000 board includes a PCI bridge, a clock generator, and an XCV2000E FPGA chip. The board is designed to allow single byte transfer to and from the FPGA chip through a dedicated address port in the PCI bus. Multi-byte transfers are possible only by redirecting the data using DMA transfer to the external memory, before being read by the FPGA chip. The XCV2000E itself is capable of running at clock speeds from 0 to 100 MHz. The board contains four memory banks, each of 2 MB, accessible to both the FPGA and any other device on the PCI bus. A single XCV2000E chip contains 19200 CLBs that roughly amounts to 2 million system gates. Each CLB in the Virtex series are divided into 2 programmable slices, therefore it is possible to program 38, 400 individual slices in the XCV2000E chip.

5.2 Fixed-Point Representation Analysis

One of the most difficult tasks in implementing an algorithm in an FPGA is dealing with precision issues to meet system requirements [148]. Typical concepts such as word size and data type are no longer valid in FPGA design, which is dominated by finer grain computational structures, such as look-up tables.

Hardware arithmetic traditionally focuses on either integer or fixed-point arithmetic representations. Due to the significant increase in resources in the latest FPGAs, it is feasible to support more complex arithmetic formats such as floating-point.

Fixed-point arithmetic is the more straightforward of the two number representations. In fixed-point representation, an implicit binary point is used to separate the integer part and the fractional part within a single data word. Fixed-point number representation facilitates implementation of most of the calculations as integer arithmetic, as little pre or post-normalization is required.

The pre and post-normalization steps used in floating-point arithmetic require the use of priority encoders and variable shifters. These components are expensive in terms of area usage and power consumption, and tend to have large combinational delays. Hence when identical range and precision are considered, floating-point operations are always more costly than fixed-point operations in terms of speed, area and power consumption.

However, there is always a tradeoff between accuracy of fixed-point representation and the hardware cost: minimum quantization error requires using wide fixed-point representations, but wider signals require larger circuits to perform mathematical operations (dividers, multipliers, adders, etc.), in addition to larger data path circuits, larger memory and higher chip-to-chip communication bandwidth.

To characterize a fixed-point arithmetic system, a commonly used representation is based on the 15.16 standard that uses the following parameters [149]:

- **WL** – World Length, total number of bits used to store the fixed-point numbers.
- **IWL** – Integral Word Length, total number of bits assigned to the integral part.
- **FWL** – Fractional World Length, total number of bits assigned to the fractional part.
- **S** – Unsigned or two's complement (signed)

Figure 5.2 presents a diagram for the fixed-point parameters used in 15.16 representation.

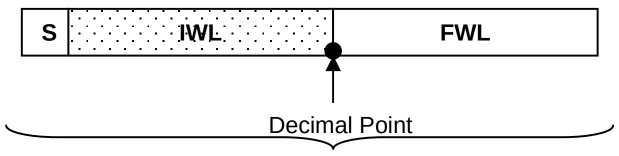


Figure 5.2 Fixed-Point representation

The range of the fixed-point numbers supported for this representation is defined in equation 5.1 [150]:

$$-2^{IWL} \leq N < 2^{IWL} \left\{ Quantization\ \ Step : 2^{-FWL} \right\} \qquad (5.1)$$

For a 32 bit word, a common choice is: IWL = 15, 1 sign bit, FWL = 16, WL = 32. This is referred as 15.16 signed representations. The minimum number that can be represented is -2^{-15} = -32768.0 (0x8000000) and the maximum number is $2^{15} - 2^{-16} \approx$ 32767.99998 (0x7FFFFFFF).

Addition and subtraction can be done in similar ways as the integer addition and subtraction. However, in order to preserve the accuracy of 15.16 signed representation, multiplication and division require a temporary accumulator with 2WL, i.e. 64 bits. This result must ultimately be stored in a fixed length 32 bit word. The least significant bits cannot simply be truncated of the end of the number since they represent the magnitude of the number, an essential part of its representation.

Scaling is a form of fixed-point operation with which the actual physical values of the original signal can be compressed, or expanded into a range suitable to the system. By suitable, it is understood that the system will be able to store the input value into its registers, and to use operations with sufficient range to hold intermediate results. It must be ensured that during all of the calculation stages, storage locations and operations suitable for the range of values required at that stage are used. Typically power-of-two are used for scaling so that this operation can be performed using shifts. In a multiplication, if both operands are scaled up by a factor N, the multiplication result is scaled up by N^2. To restore the original scaling, the result has to be divided by N.

The standard just presented has been adjusted to the current architecture requirements. In this work gray level images are used, each pixel is represented using binary numbers ranging from 0 to 255 in decimal to cover all the possible pixel values presented in the input images therefore 8-bit internal registers are needed for data management. Certain window-based algorithms, such as Gaussian filtering, may present fixed-point mask coefficients. Considering a Mean Squared Error (MSE) of 5.6, 8-bit internal registers are needed for data management. Furthermore negative values may be present in the mask coefficients, if an operation between these values and the image pixels is performed the result is also a negative value, therefore an extra bit must be taken into account for sign representation, resulting in a requirement of 9-bit for the integer part of the representation.

Based on these characteristics the 8.8 standard has been established for the present architecture. The word length used is of 17 bits considering that: IWL = 8, 1 sign bit, FWL = 8, WL = 17. In this particular case the minimum number that can be represented is $-2^8 = -256.0$ (0x100), the maximum number is $2^8 - 2^{-8} \approx 255.99609375$ (0xFF) and the resolution is $2^{-8} = 0.00390625$.

In order to implement operations presented in Table 3.2 in hardware, the PEs has been designed to support the 8.8 standard. To manage the fractional part of the number the value has been scaled by 256, this is achieved by using shift registers. Consequently operations can be performed like 8-bit integer operations using a 25 bit register for intermediate results. In this way it is possible to take advantage of the general optimizations made to the ALU since no additional hardware logic is required. The disadvantage of using this schema is that only a limited range of values can be represented, as a consequence this representation is susceptible to common numeric computational inaccuracies.

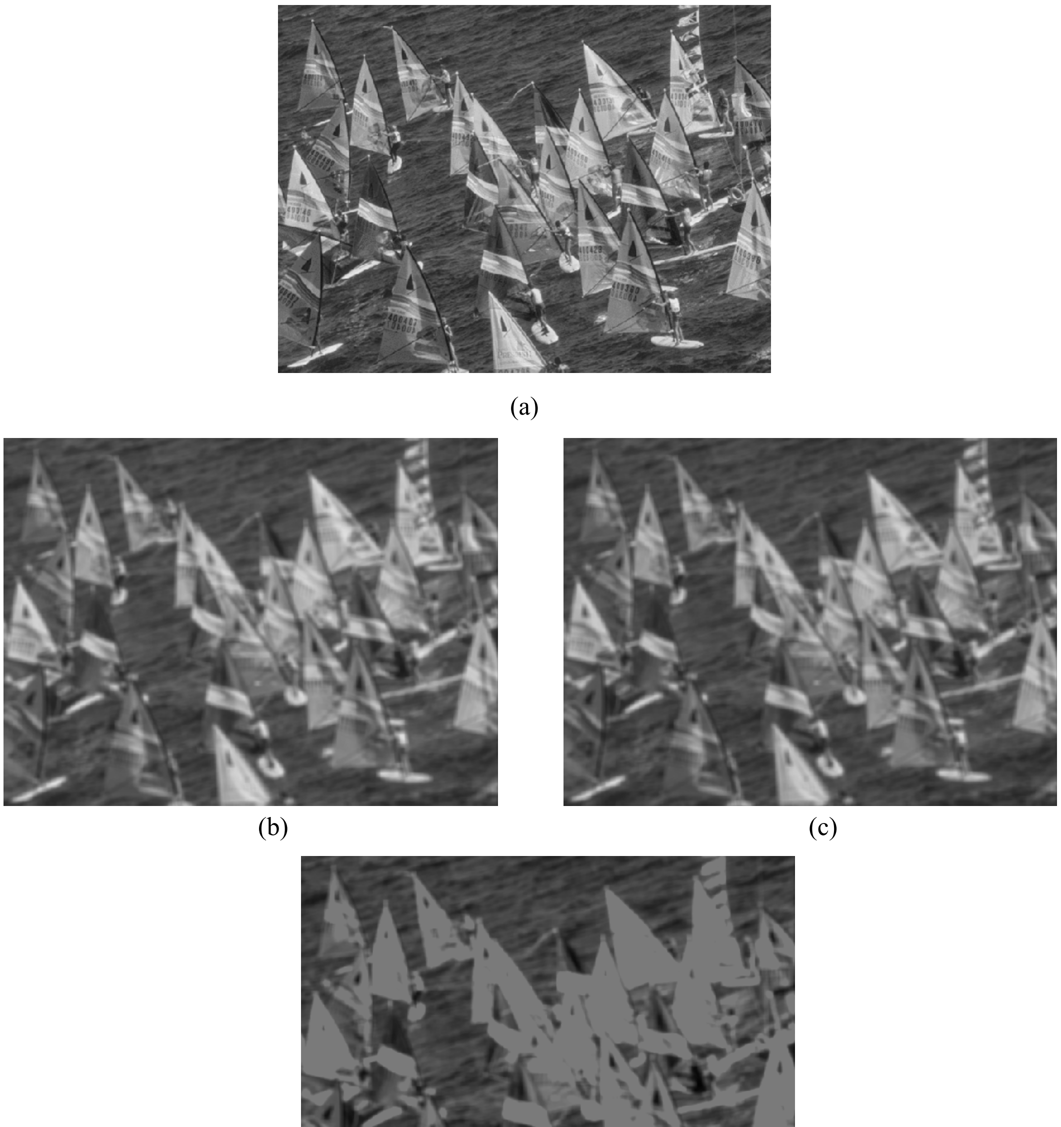
For example, consider a mask coefficient from a Gaussian filter that is represented in two's complement binary format. To keep the example simple, a positive number is considered. To encode 0.625, the first step is to find the value of the integer bits. The binary representation of the integer part is 000000000. The fractional part of the number is represented as 0.625×2^n where *n* is the number of fractional bits used for representation. In this case according to the standard established, *n* is 8. If the number is scaled by 2^8=256, the fractional part transforms to 0.625×256 = 160, whose representation is 10100000 thus, the binary representation for 0.65 is 0 0000 0000 1010 0000.

Using the scaled mask coefficients and the available input image values, it is possible to perform operations without modifying the PE basic structure; the only extra element is an output shift register that executes the required division by 256 when the final result is obtained.

The fractional part of the numbers used in the algorithms has been adjusted to an 8-bit format therefore some information representing the magnitude of the number has been truncated consequently an error in the final result is introduced due to this quantization process. The measurement of this error is important in order to keep an acceptable margin according to the constraints of the algorithm.

A first approach consists in creating an absolute error image by subtracting the obtained image processed in the architecture from the image obtained by simulation in Matlab and then taking the absolute value. In figure 5.3 it can be observed the result for this process using a mean filter. The error image provides a visual aid to identify the amount of image distortion.

In order to provide a numerical measure of overall distortion in the image another approach is to compute the mean squared error over the picture. This is a common metric used to determine the error allowed for an application. The 8.8 data representation used in this thesis has been defined based on this metric.


(a)

(b)

(c)

(d)

Figure 5.3 Error image for a mean filter: (a) Original Image, (b) Output image processed by the architecture, (c) Output image from Matlab simulation, (d) Error image

The MSE is the cumulative squared error between two images. The mathematical formula for the MSE is defined in equation 5.2 [151]:

$$MSE = \frac{1}{MN}\sum_{y=1}^{M}\sum_{x=1}^{N}\left[I(x,y) - I'(x,y)\right]^2 \qquad (5.2)$$

Where I(x, y) is the original image, I'(x, y) is the approximated version of the image and M and N are the dimensions of the images. A lower value for MSE means a smaller error.

Table 5.1 shows the MSE for different truncation errors using a pair of 640×480 gray level images. As can be observed, after certain value the error decreases slowly. It doesn't matter if more bits are added for data representation using a longer shift register the value of MSE is not significantly affected, therefore a shift by 256 has been chosen, achieving a resolution of 0.00390625 for data representation. The MSE shown in the output image is around 5. In comparison with the integer version of the architecture, the growth in area is around 2%.

Table 5.1 Mean squared error

Value of shift used	**MSE**
64	28,6052
128	28,6049
256	5,6267
512	5,6264

Using the 8.8 representation, according to the Table 5.1 the measured MSE is 5.6. In Figure 5.4 a new error image is presented. This image is obtained subtracting the obtained image processed in the architecture using 8.8 representation from the image obtained by simulation in Matlab. As can be observed the error has been reduced in comparison with the image in Figure 5.3.

Figure 5.4 Error image for a mean filter using 8.8 representation

Figure 5.5 shows the MSE's behavior for images of 640×480. If the number of bits for data representation changes from 7 bits to 8 bits the error reduction is about 3.5%. In the figure there is a region where MSE remains approximately constant; from 8 bits to 10 bits the error is not reduced. From this point, once again the error diminishes as the number of bits used increases.

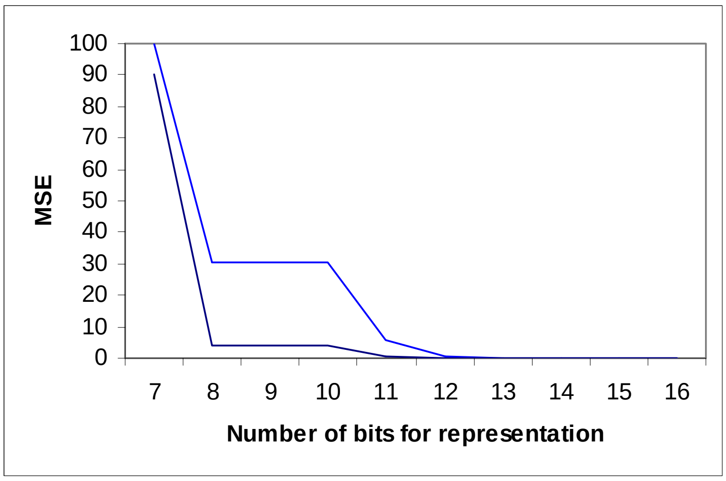


Figure 5.5 Mean squared error vs. width of inputs

In order to choose the appropriate data representation, this behavior has been taken into account. The intention is to maintain a low error value which implies the use of a reasonably number of bits, maintaining a reduced amount of the area occupied. Therefore it is necessary to find a good tradeoff between these two parameters.

According to Figure 5.5 an acceptable error can be chose in the range of 8-10 bits due to a bigger number of bits implies an important growth in area, and a value below implies a great amount or error.

In order to have a reference of the area behavior, a graphic of the cost of implementing arithmetic multipliers and dividers versus the input width of the multiplier or divider measured in number of LUTs is presented in Figure 5.6. As can be observed the area increases approximately with the square of the input width.

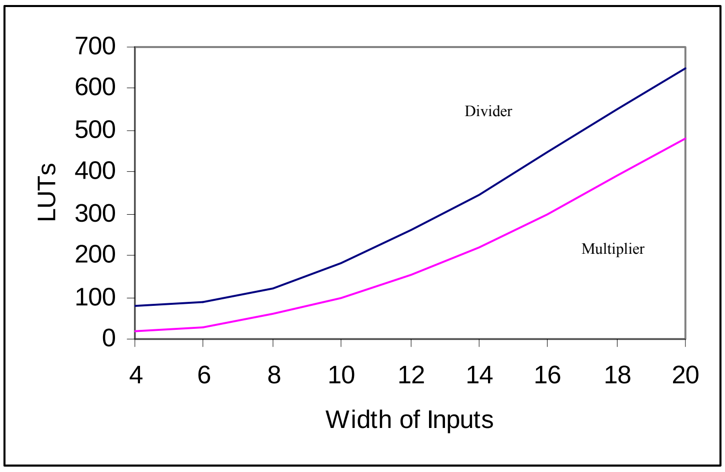


Figure 5.6 Hardware cost for arithmetic operation

Considering the observations made from Figures 5.5 and 5.6, 8 bits have been chosen to represent the fractional part of fixed-point numbers achieving a good tradeoff between accuracy and area occupancy with the 8.8 representation. The number of bits selected is enough to avoid the occurrence of overflow.

5.3 Implementation and Synthesis Results

In this section, synthesis results for all the modules implemented in the architecture are presented.

Every module has been described using the hardware description language Handel-C and for the implementation it has been used FPGA technology. The architecture has been synthesized to a XCV2000E-6 Virtex-E FPGA with the Xilinx Synthesis Technology (XST) tool and placed and routed with Foundation ISE 7. The flow diagram used in order to test the architecture is shown in Figure 5.7.

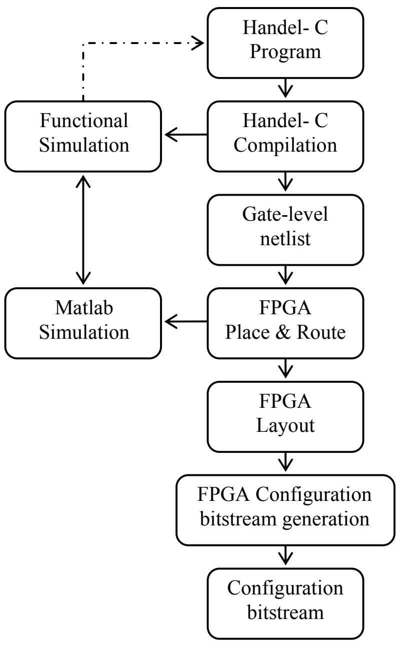


Figure 5.7 Flow diagram to test the architecture

As can be observed, to evaluate the hardware architecture, the process starts with the high level description of the architectural modules using Handel-C DK4. The functionality of the design is then verified using the Handel-C simulator environment. Once satisfactory simulation results have been achieved the Handel-C compiler can be used to generate the FPGA gate-level netlist. The netlist format created by the Handel-C compiler is EDIF.

Target technology-specific placement and routing tools are used next to map the gate-level netlist generated by the Handel-C tools into the targeted FPGA device. In this implementation ISE Foundation from Xilinx has been used for this purpose. The FPGA placement & routing tools will produce a final layout of the FPGA design. From this layout, the FPGA tools can also generate an FPGA configuration bitstream, which will be used to program the target FPGA device. Finally, in order to verify the post-place and route results, they are compared to the results obtained by simulation using Matlab.

With the purpose of demonstrating the correct operation of the architecture, a window mask of 7×7 coefficients and a systolic array of 7×7 PEs are used.

5.3.1 Processing Element Implementation

A detailed description of the internal structure of one PE is shown in Figure 5.8. The PE is composed of two main elements, the ALU that provides the functions of multiplication, addition, subtraction, and absolute value and the Accumulator that implements the local reduction functions of accumulation, maximum, minimum and absolute value. Each PE has two operational inputs, pixels from the input image and coefficients from the window mask denoted by P and W, respectively in the Figure.

The ALU includes a multiplier, an adder-subtracter and a Sum of Absolute Differences (SAD) module. The Accumulator module includes an accumulator, and a maximum-minimum computation module. The operation performed by these modules can be configured by a control word selected by the user according to the algorithm to be implemented. The PE has two output signals, the partial result of the window operation and a delayed value of a window coefficient that is transmitted to its neighbor PE. The PE internally has two registers; the first one has 8-bits width and stores the window mask coefficient temporally to transmit it to the next PE after one clock cycle. The second is a 25-bits shift register that is used to normalize partial results when a fixed-point operation is performed.

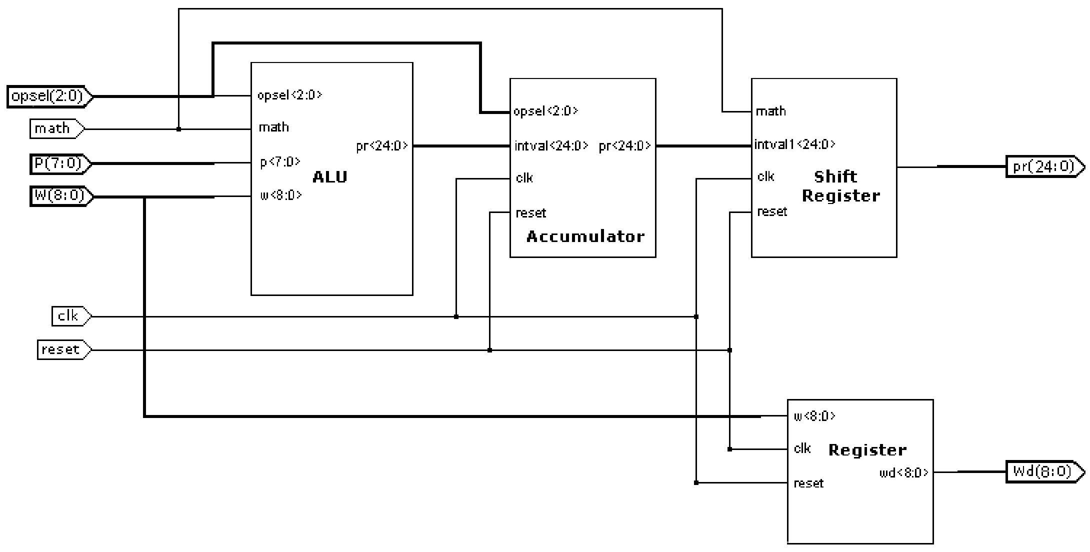


Figure 5.8 Internal structure of the PE

The signal description of this architectural block is presented in Table 5.2.

Table 5.2 Signal description for PE module

Signal	Direction	Width	Description
p	In	8	Pixel of the input image
w	In	9	Coefficient of the window mask
opsel	In	3	Control word to configure the PE according to application
math	In	1	Data type selector: Integer/Fixed-point
reset	In	1	Reset signal for the PE block
clk	In	1	Main clock
P_R	Out	25	Partial result for an operation
W_d	Out	9	Output for delayed value of the window mask

For a single processing element that is performing a filtering operation the synthesis results are presented in table 5.3.

Table 5.3 Technical data for the PE

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 µm 6-layer metal process
Number of Slices	720 out of 19200
Number 4 input LUTs	1,318 out of 38,400
Number of Flip Flops	569 out of 38,400
Overall % occupancy	3 %
Internal data buses for ALUs	8 bits for fixed-point operations
Power Consumption	550 mW

5.3.2 Router Implementation

The Router is responsible for all data movement in and out of the systolic array as well as interfacing processing modules to external memories. The internal structure of the router is shown in Figure 5.9.

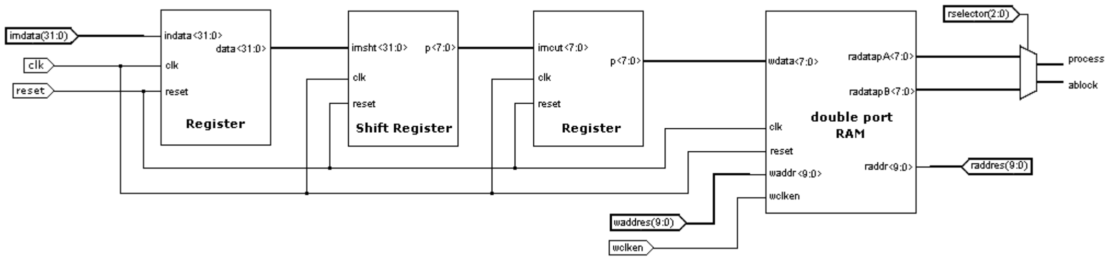


Figure 5.9 Internal structure of the Router

This block is composed of an address generator that produces the address to access the input image memory. A shift register that parallelizes the read pixels to store them in a buffer implemented with double port BlockRAM memories. The generator produces as well the addresses to access the buffer that stores the input image rows;

this buffer allows the parallel access to data. The signal description of the Router module is presented in Table 5.4.

Table 5.4 Signal description for Router module

Signal	Direction	Width	Description
imdata	In	32	Pixel of the input image
waddres	In	10	Address where input pixels are stored inside the BlockRAM memory
wclken	In	1	Write enable signal
rselector	In	2	Data flow selector: To current processing block/next processing block
raddres	In	10	Address to read data out of memory
reset	In	1	Reset signal for the Router block
clk	In	1	Main clock
radatapA	Out	8	Output data read from port A of the memory
radatapB	Out	8	Output data read from port B of the memory

The synthesis results for this module are presented in Table 5.5.

Table 5.5 Technical data for the Router

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of Slices	330 out of 19200
Number 4 input LUTs	397 out of 38,400
Number of Flip Flops	221 out of 38,400
Overall % occupancy	1 %
Power Consumption	800 mW

The address generator is composed of counters, adders, and registers. This block is in charge of scanning the input image memory, producing the addresses to read the input pixels and generating the input addresses to the BlockRAM memories that cache the corresponding input image rows. This module is presented in Figure 5.10.

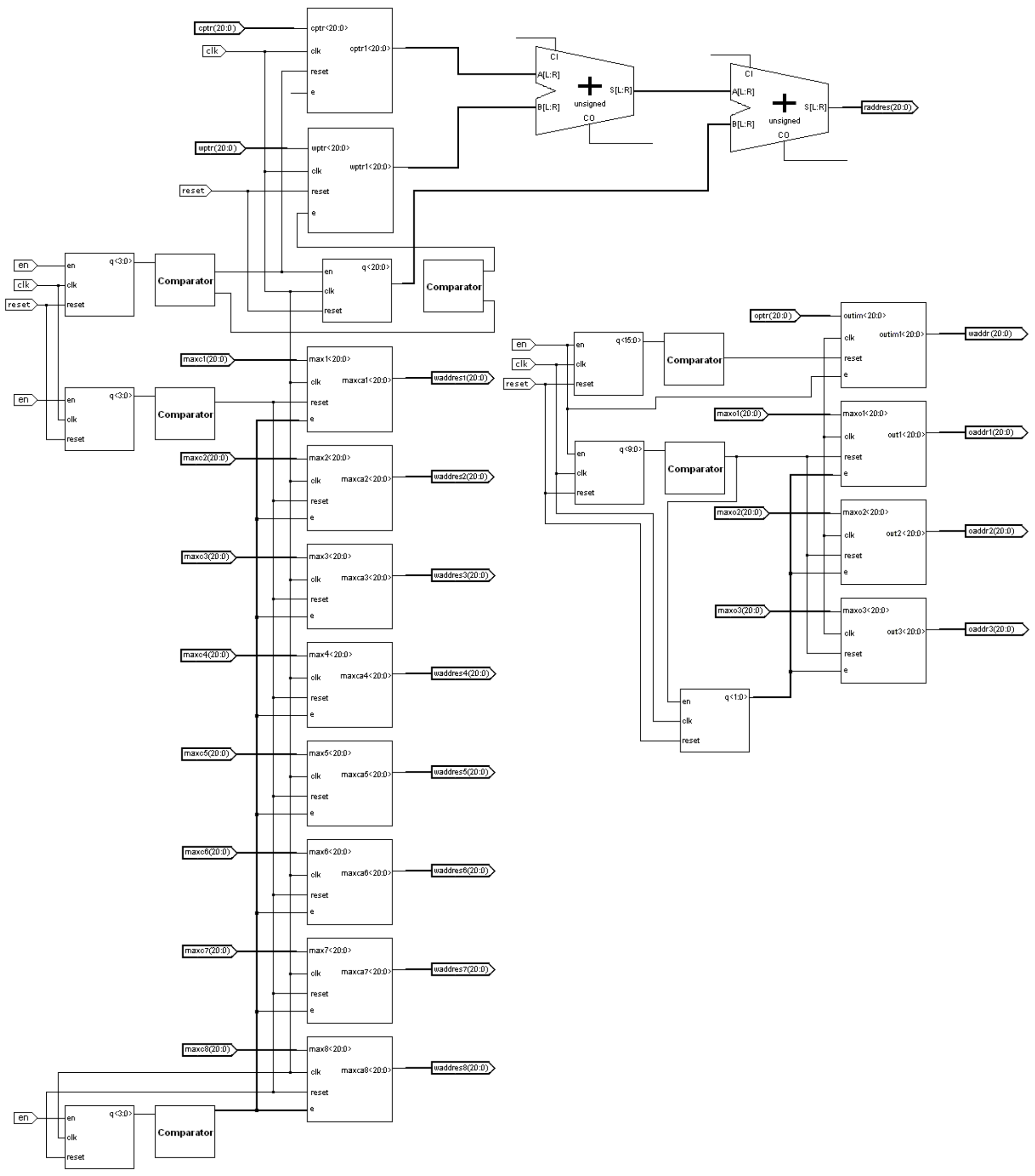


Figure 5.10 Internal structure of the address generator

The signal description of the address generator module is presented in Table 5.6.

Table 5.6 Signal description for address generator module

Signal	Direction	Width	Description
cptr	In	21	Pointer to the initial column address in the input memory
wptr	In	21	Pointer to the initial row address in the input memory
maxc1-8	In	21	Pointers to the memories constituting the input buffer
maxo1-3	In	21	Pointers to the memories constituting the output buffer
optr	In	21	Pointer to the initial column address in the output memory
en	In	1	Enabler to start the address generation process
reset	In	1	Reset signal for the address generator block
clk	In	1	Main clock
raddres	Out	21	Address generated to direct the input image memory
waddres1-8	Out	21	Address generated to direct the input image buffer
waddr	Out	21	Address generated to direct the output image memory
oaddr1-3	Out	21	Address generated to direct the output image buffer

In the first clock cycle the pointer to the input memory *cptr* is initialized to point out the first memory locality. The input image is scanned in a column-based order; in each memory locality four pixels from the input image are stored and every clock cycle the pointer indicates the memory address being read. The pointer is incremented by the number of columns in the input image to read the next image row, until eight rows are read. Afterwards, the column counter's accumulator is set to its initial value and the process is started again. This column based value is added to the counter of columns and the base address that provides the reference row in the input image is set to its new position using an increment of $N \times C$, where N is the number of columns in the input image and C is the number of columns processed in parallel.

The data read from the input memory are stored in the eight BlockRAMs constituting the image buffer. A group of pointers, *maxc1* to *maxc8*, select the buffer row being filled. Every clock cycle the pointers are incremented to go over all the buffer memories localities storing in parallel an image pixel in each buffer row.

When the first buffer row is being filled, the pointer that selects the buffer row *maxc1* is selected to store the new data that are being obtained. Once the first buffer row is full the current input memory address in *cptr* is increased by a factor that advances the pointer to the next input image row and these pixels are read, then the buffer pointer *maxc2* is selected to store the read data and the process continues this way until the eighth row is reached with *maxc8*. Afterwards the process is repeated, setting the buffer pointer to the first row *maxc1* once again, in this way a circular pattern is achieved. This cycle continues until the whole input image is completed.

The address generator is also in charge of generating the addresses to store the processed pixels in the output memory. When pixels have been processed, they are stored in the output buffers which are constituted by three BlockRAM memories directed by *maxo* pointers. On each clock cycle, the address in the first buffer is incremented until the final address is reached, then the pointer is set to the second memory buffer while data stored in the first buffer are arranged and sent to the output memory.

When the end of the second buffer memory is reached, the third buffer is selected to store processed pixels as data from second buffer is sent to the output memory. When the third buffer is full, then the first one is used again to complete a circular pattern. The addresses in the output image memory are incremented one by one until all the processed pixels have been stored. The synthesis results for the address generator module are presented in Table 5.7.

Table 5.7 Technical data for the Address generator

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of Slices	112 out of 19200
Number 4 input LUTs	55 out of 38,400
Number of Flip Flops	55 out of 38,400
Overall % occupancy	1 %
Power Consumption	200 mW

5.3.3 2D systolic architecture

For a systolic array of 7×7 PEs, the logic resource utilization using as main parameters the number of slices, the number of flip-flops, the number of look-up tables, and the number of BlockRAMs as well as some other features for the architecture are shown in Table 5.8.

Table 5.8 Technical data for the 2D systolic array

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	2,855 out of 19200
Number 4 input LUTs	5,241 out of 38,400
Number of Flip Flops	1,628 out of 38,400
Number of Block RAMs:	11 out of 160
Overall % occupancy	37 %
Clock frequency	66 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	5.9 GOPs
Power Consumption	1.45 W

The hardware resource utilization for the 2D systolic array is about 37% of total logic available in the FPGA. This area occupation allows the possibility of repeating the architecture blocks inside the same FPGA more than once, to execute different window operators at the same time. Another possibility to take advantage of the reduced area required for the architecture consists in the implementation of larger arrays using masks from 9×9 to 16×16 coefficients achieving a reduced degradation in current performance.

In order to test the architecture, different window coefficients have been generated and the functionality of the architecture has been selected specifying a control word for a configuration register through a software interface. To validate the functionality, images and arrays of different size have been used, however results only for a 640×480 gray-level image and window masks of 7×7 are shown.

Considering the characteristics of the 640×480 images used, the size of the internal buses and registers has been defined. For gray level images each pixel is represented using 8-bit unsigned binary number format ranging from 0 to 255 in decimal to cover all the possible pixel values presented in the input images. The Mask's coefficients may represent negative values. These values are stored in internal BlockRAM memories. Virtex-E structure, allows implementing 8-bit or 16-bit wide memories. Mask coefficients require 8-bit representation for their 256 possible values and one extra bit for sign representation, therefore 9-bit are used for full representation and 16 bit wide memories are used for hardware implementation.

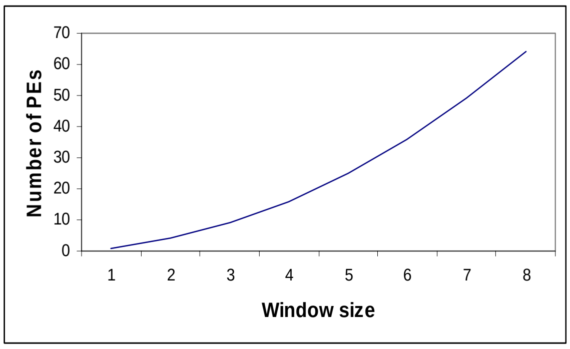
If these number formats are considered, when the basic operations in Table 3.2 are performed, the maximum size for internal register is obtained. Multiplication is the operation that requires the biggest amount of bits for data representation so it has been taken as reference to define internal architecture register requirements. Multiplication of two operands with 8 and 16-bit respectively requires a minimum of 24 bits and one extra bit is added for signed number representation. Based on this fact, internal register and buses have been adapted to fulfill the requirements of this arithmetic operation.

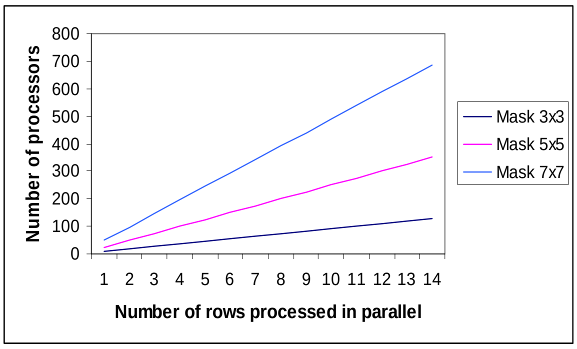
For this case the clock frequency reported by the synthesis tool is above 66 MHz; therefore a single frame is computed in around 5 ms. equivalently, the proposed architecture is able to process about 200 gray-level images per second, which is better than the previously reported in the literature [141, 152, 153]. Achieving an improved performance with comparable resource utilization under real time constrains.

5.4 Performance Analysis

The processing time and the degree of parallelism are important parameters to be taken into account to determine the architecture performance. The hardware architecture throughput can be expressed in terms of the number of images processed per second; therefore the FPGA synthesis results and Equations (5.3) and (5.4) were employed to plot some performance graphs that help to determine the best tradeoff between these parameters to achieve the highest throughput.

Image processing applications do not imply only the application of a window-based image operator but a sequence of different image operators, for this reason it is important to reduce the processing time. One possibility is to increase the number of rows processed in parallel. In Figure 5.11 (a) it can be observed that the growth in the number of PEs for different windows size is quadratic. If this schema is replicated for multiple rows processed in parallel, the growth in the number of PE is linear as shown in Figure 5.11 (b). In both cases the amount of area used grows in the same proportion which is a disadvantage for the implementation. According to graphs in Figure 5.11, 49 PEs have been selected. This number of PEs allows the management of windows of 7×7 which is enough for some applications and offers a good commitment between area and throughput as can be observed in Figure 5.12, where throughput *vs*. the number of rows processed in parallel is shown. It can be observed that even if the quantity of rows processed in parallel is increased, the gain in speed is not quite significant; consequently a good choice is a throughput of 1 for 49 PEs.





(a) (b)

Figure 5.11 Growth in the number of processor: (a) PEs required per window size, (b) PEs required per number of rows processed in parallel

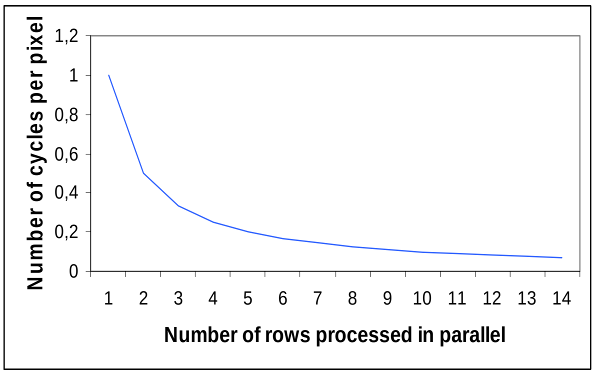


Figure 5.12 Performance vs. Parallelism

From the synthesis results, the most important factor to take into account is the maximum clock frequency achieved. One challenge in hardware design is to maximize this parameter. As can be observed from Figure 5.13; the bigger the frequencies of main clock, the better the performance in the architecture. However after a certain value even if the clock frequency is increased, the processing time reduction is very little.

For real time operation, the common standard is 30 frames per second. This goal is achieved if the frequency is chosen in values where the slope of the graph in Figure 5.13 is around 45º in Figure 5.13. In this case a frequency of 66 MHz. is enough to process about 200 images per second; this frequency is around the value reported by the synthesis tool. Thus the architecture provides enough computational power.

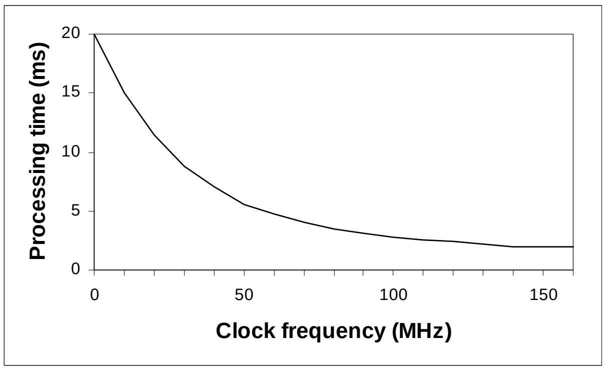


Figure 5.13 Throughput

The hardware architecture presented in the previous section is dedicated to window-based operation. The main objective during architecture conception is to achieve real time operation; therefore an analysis in performance parameters is required. In this section the processing time of an input image in terms of architectural parameters is computed.

The time required to process an input image with a window-based operator is composed of two main times; the *latency time* and the *parallel processing time*. The latency time is measured between the activation of the first PE and the activation of the last one. The latency is significant when the 2D systolic array starts the processing of a new set of rows of the input image. The total time required to initialize full pipeline operations of the parallel modules at the beginning of row processing is summarized in Equation 5.3.

$$\tau_l = WindowSize \;\; x \;\; \frac{1}{f} \;\; x \;\; \frac{N}{C} \qquad (5.3)$$

Where *WindowSize* corresponds to the size of the mask used in the computations, *f* is the frequency of the main clock, *N* is the width of the input image and *C* is the number of columns processed in parallel.

The parallel processing time is the time when all the PEs in the 2D systolic array are working in parallel through the whole image processing without considering the latency at the start of the processing. For an M×N image, the processing time is given by Equation (5.4):

$$\tau_p = M \;\; x \;\; N \;\; x \;\; \frac{1}{f} \qquad (5.4)$$

The overall time needed to process an M×N image is given by the addition of the preceding times [141]:

$$T = \tau_l + \tau_p \qquad (5.5)$$

For the values considered, a total time consumed of approximately 5 ms is obtained; therefore about 200 images can be processed per second.

Performance can be also measured based on the number of elementary operations that the system can perform per second. For window operators only the operations contributing to the computation of a window result are considered, in this case to determine the performance, the operations involved in convolution are taken as basis for calculation. The total amount of operations Op_T per second is defined in Equation (5.6):

$$Op_T = (Op_f)(Num_f) \tag{5.6}$$

Where Op_f is the total number of operations performed in a frame and Num_f is the total number of frames processed in a second.

For a 640×480 image and a window mask of 7×7, 29.7 operations are required per frame. Whit the throughput achieved by the presented architecture, this number of operations is accomplished to operate in real time.

5.5 Scalability analysis

In order to estimate the scalability and the level of improvement in performance of the architecture, it has been synthesized to different technologies. The Virtex E (xcv2000E), Virtex-II (xc2v2000), Spartan III (xc3s2000), Virtex II-Pro (xc2vp2) and Virtex IV (xc4vlx200) families were considered for the filtering application using an array of variable size and a 640×480 input image. The obtained results are plotted in Figure 5.14 for the amount of area consumed.

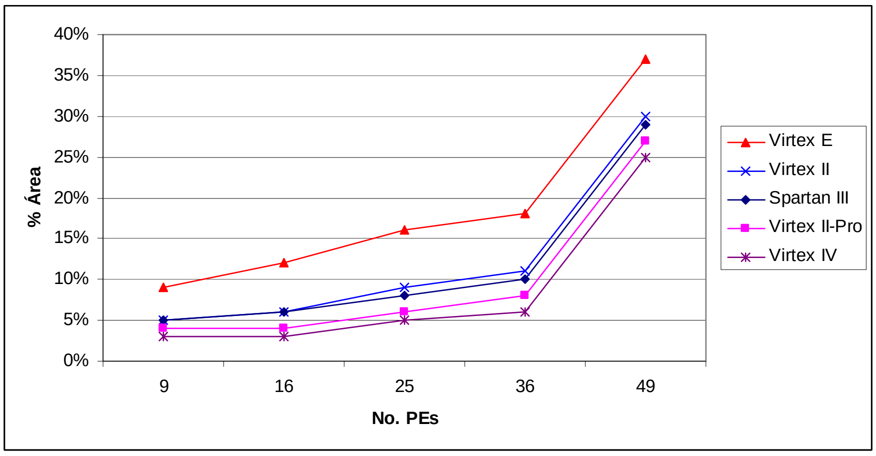


Figure 5.14 Area occupancy for different FPGA families

As can be observed from the Figure, there is a reduction in area occupied by the architecture as the technology improves. This is basically due to the reduction in the size of the transistors used to implement the FPGA devices and to the existence of embedded multipliers in most of the advanced FPGA families.

Figure 5.15 shows the number of CLBs consumed by the architecture synthesized for the FPGA families considered. This parameter is another reference for resource occupation comparison.

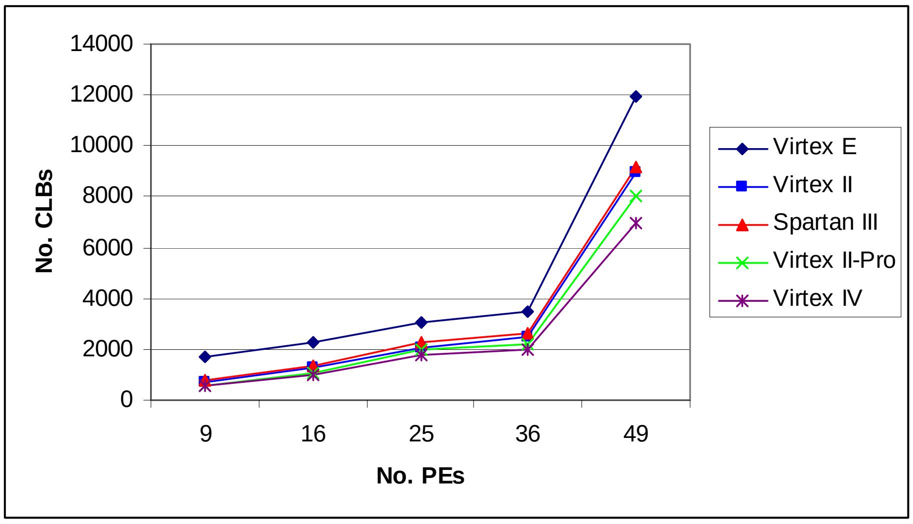


Figure 5.15 Number of CLBs for different FPGA families

The implementation of the architecture using embedded multiplier requires less hardware resources than the version with the distributed logic multipliers due to the reduction of the complexity in the interconnection inside the architecture. As a result of this area reduction an enhancement in the architecture performance is expected with the improvement of the technology.

Figure 5.16 presents the frequency achieved by the architecture for the FPGA families considered in the analysis. The plot was obtained using an array of variable size and an input image of 640×480. As can be observed, Virtex II-Pro and Virtex IV provide the higher performance due to the improvements in their internal structure.

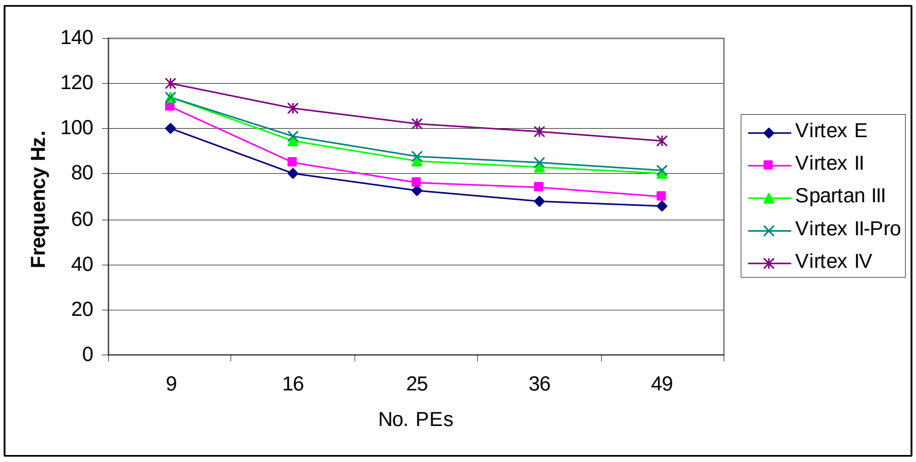


Figure 5.16 Frequency for different FPGA families

Considering these results, a higher throughput is expected if the architecture is scaled to a newer technology. However the performance achieved using an xcv2000E FPGA fulfils the real time requirements for the applications proposed in the present work.

5.6 Architecture Discussion

The main characteristics and the operation of the architecture was presented in previous sections; nevertheless some features are highlighted here in order to emphasize the advantages of the current schema in comparison with previous approximations presented in the literature.

The main module of the architecture is arranged as a systolic array. This structure presents several benefits to implement applications that demand high computational power. Systolic arrays offer flexibility and provide a high throughput. This thesis addresses the implementation of a hardware architecture for image processing where some enhancements have been added to extend the systolic array functionality and to improve high performance.

One way to strengthen the capacity of the architecture is to provide it with the capability of processes chaining. In order to achieve this goal, the systolic architecture has been coupled with other elements such as memory buffers and router blocks. The interaction of these components has driven the improvement of four features:

- **Modularity**: This is a fundamental feature that has been adhered to the architecture. Replicating the basic schema presented in Figure 4.3 it is possible to configure individual blocks of the system making them customizable. In this way it is possible to process different algorithms at the same time which represents a remarkable difference between the present architecture and previous approximations in the literature where generally a single algorithm is processed. This modular structure is an advantage when the output of an algorithm depends on the processing of the information along several stages; hereby it is possible to re-use information along subsequent processing stages where the output of one block corresponds with the input of the following one. Modularity improves performance due to all the processing blocks located inside the same integrated circuit. The disadvantage is that this can be applied to FPGAs of great capacity. Dynamic reconfiguration techniques are an alternative to improve the hardware resource utilization due to the partial reconfiguration of the FPGA during execution. In this way customization can be performed in real time reducing area occupancy.
- **Communication**: The routers developed in the architecture for a single-chip platform support complex communication. They are in charge of all inter-modular data movement tasks. Most of the architectures presented in the

literature count with limited data movement through a fixed structure, so the router elements provide major flexibility and capacity for processes chaining inside the FPGA. The routers reduce the bandwidth necessary for the communication among processes, which represents another advantage over previous systems.

- **Memory**: The amount of memory available in an FPGA based system is usually limited therefore it is necessary a smart management of existing resources. Off-chip memory is used to store input and output images, reading and writing data is a time consuming process therefore the number of accesses to memory have been reduced in the presented architecture. In comparison with previous systems, data are read only once during processing; moreover data read are reused via small internal buffers that increase data parallelism. During processes chaining, image buffers allow to share and to reuse data between processing blocks. In the presented architecture an efficient use of memory has been achieved resulting in reduced area occupancy.
- **Data flow**: In the presented architecture data movements are systolic flow. In each cycle a single pixel value is transported across PEs in the array as in most other architectures previously presented. Router elements add extra flexibility to the management of data. This creates an image flow bus inside the architecture that reinforces the architecture capacity for process chaining. This is an advantage with regard to other presented schemas. An additional outstanding feature in the proposed architecture is the presence of internal buses dedicated to the management of configuration and control parameters. These buses constitute a means for architecture customization.

Taking into account the characteristics above mentioned it is deduced that the proposed architecture presents several improvements with respect to previous works. The architecture offers a high potential to implement complex algorithms via processes chaining which is an approach scarcely explored in literature.

The architecture has probed to be a good tool for implementation of window-based algorithms achieving a good tradeoff between performance and area occupancy under real time conditions. However a performance improvement can be obtained either by optimizing the design mapped onto the FPGA or by employing FPGAs with better technology.

Chapter 6

Architecture Applications

In this section, the results for some low-level application are presented. To validate the hardware platform functionality and flexibility, convolution, filtering, matrix multiplication, pyramid decomposition and template matching algorithms have been mapped into the systolic array.

All the output images presented in this section correspond to the data obtained directly from the FPGA board prototype and they were compared against the software implementations on a general purpose computer. In all the test cases, the correct results were obtained and validated.

6.1 Convolution

Convolution is a simple mathematical operation which is fundamental to many common image processing operators [154]. Convolution is a way of multiplying together two arrays of numbers of different sizes to produce a third array of numbers. In image processing the convolution is used to implement operators whose output pixel values are simple linear combinations of certain input pixels values of the image. Convolution belongs to a class of algorithms called spatial filters. Spatial filters use a wide variety of masks, also known as kernels, to calculate different results, depending on the desired function

The basic idea is that a window of some finite size and shape is scanned over an image. The output pixel value is the weighted sum of the input pixels within the window where the weights are the values of the filter assigned to every pixel of the window. The window with its weights is called the convolution mask. An important aspect of convolution algorithm is that it supports a virtually infinite variety of masks, each with its own feature. The concept for this operator is shown in Figure 6.1.

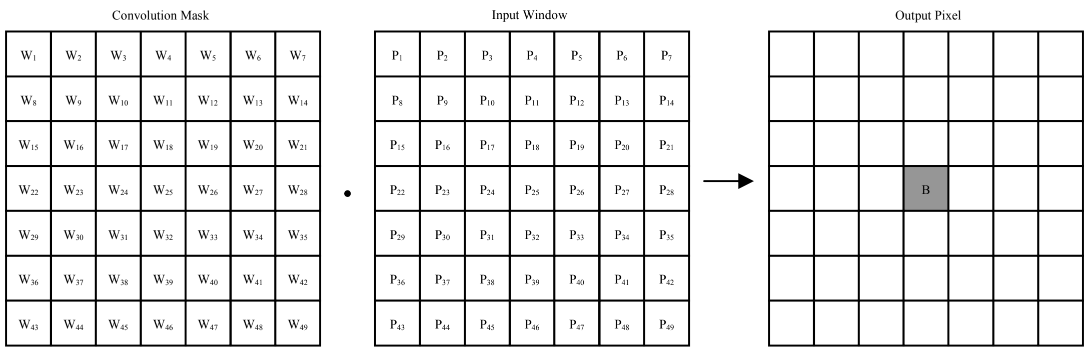
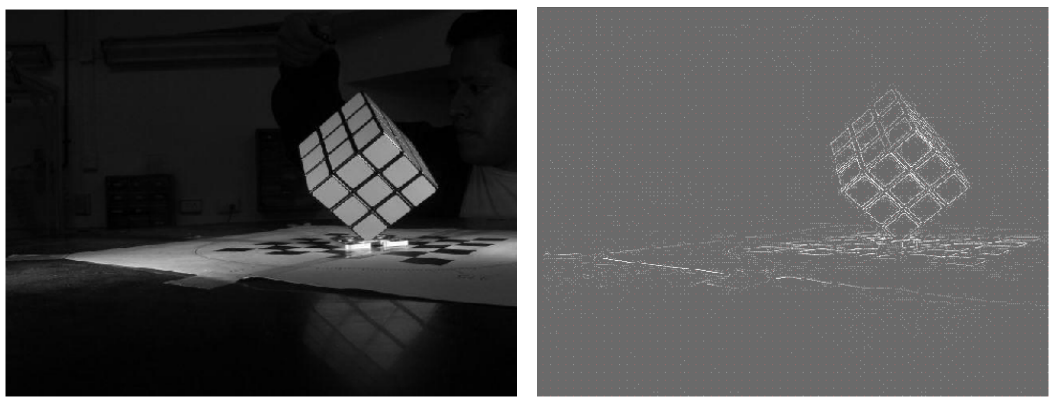

Convolution Mask
W_1 W_2 W_3 W_4 W_5 W_6 W_7
W_8 W_9 W_{10} W_{11} W_{12} W_{13} W_{14}
W_{15} W_{16} W_{17} W_{18} W_{19} W_{20} W_{21}
W_{22} W_{23} W_{24} W_{25} W_{26} W_{27} W_{28}
W_{29} W_{30} W_{31} W_{32} W_{33} W_{34} W_{35}
W_{36} W_{37} W_{38} W_{39} W_{40} W_{41} W_{42}
W_{43} W_{44} W_{45} W_{46} W_{47} W_{48} W_{49}
Input Window
P_1 P_2 P_3 P_4 P_5 P_6 P_7
P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14}
P_{15} P_{16} P_{17} P_{18} P_{19} P_{20} P_{21}
P_{22} P_{23} P_{24} P_{25} P_{26} P_{27} P_{28}
P_{29} P_{30} P_{31} P_{32} P_{33} P_{34} P_{35}
P_{36} P_{37} P_{38} P_{39} P_{40} P_{41} P_{42}
P_{43} P_{44} P_{45} P_{46} P_{47} P_{48} P_{49}
Output Pixel
B

Figure 6.1 Convolution concept

In order to test the correct operation of the convolution, the algorithm has been executed in the FPGA board. The resultant image obtained is observed in Figure 6.2. for Laplacian operator.



(a) Original image *(b) Filtered Image*

Figure 6.2 Output for convolution using Laplacian operator

The architecture has been set to process a 640×480 input image and a 7×7 window mask. The processing elements are configured to perform the scalar function of multiplication and the reduction function of accumulation over integer mask coefficients. The technical data of post-place and route for the convolution architecture are shown in Table 6.1.

Table 6.1 Technical data for the convolution

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	11,969 out of 19,200
Number 4 input LUTs	5,241 out of 38,400
Number of Flip Flops	1,628 out of 38,400
Number of Block RAMs:	13 out of 160
Overall % occupancy	62 %
Clock frequency	66 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	5.9 GOPs
Power Consumption	2.017 W

To illustrate the convolution algorithm using fixed-point numbers, a Gaussian convolution mask has been chosen. In this case a 640×480 input image is used and coefficients in the window mask have a four decimal number representation. Inside the architecture 8.8 representation is used to perform fixed-point operations. The output image obtained from the FPGA implementation is shown in Figure 6.3. The post-place and route results correspond with the obtained in table 6.1.

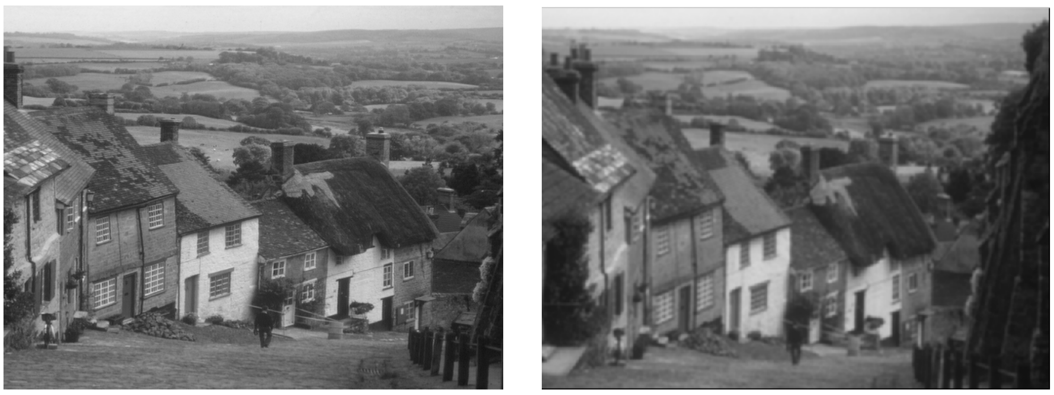

(a) Original image *(b) Filtered Image*

Figure 6.3 Output for convolution using Gaussian operator

6.2 Filtering

Digital images can be processed in a variety of ways. The most common one is called filtering and creates a new image as a result of processing the pixels of an existing image. Each pixel in the output image is computed as a function of one or several pixels in the original image, usually located near the location of the output pixel.

These algorithms are applied in order to reduce noise, sharpen contrast, highlight contours, and to prepare images for further processing such as segmentation [155]. These algorithms can be divided in linear and non-linear where the former are amenable to analysis in the Fourier domain and the latter are not.

6.2.1 Median Filtering

A Median filter is a non-linear digital filter which is able to preserve sharp signal changes and is very effective in removing impulse noise (or “salt and pepper noise”)

[156]. An impulse noise has a gray level with higher low that is different from the neighborhood points. Linear filters have no the ability to remove this type of noise without affecting the distinguishing characteristics of the signal; median filters have remarkable advantages over linear filters for this particular type of noise. Therefore median filter is very widely used in digital signal and image/video processing applications [157].

A standard median operation is implemented by sliding a window of odd size over an image. At each window position the sampled values of signal or image are sorted, and the median value of the samples is taken as the output that replaces the sample in the center of the window as shown in Figure 6.4 for a window mask of 7×7.

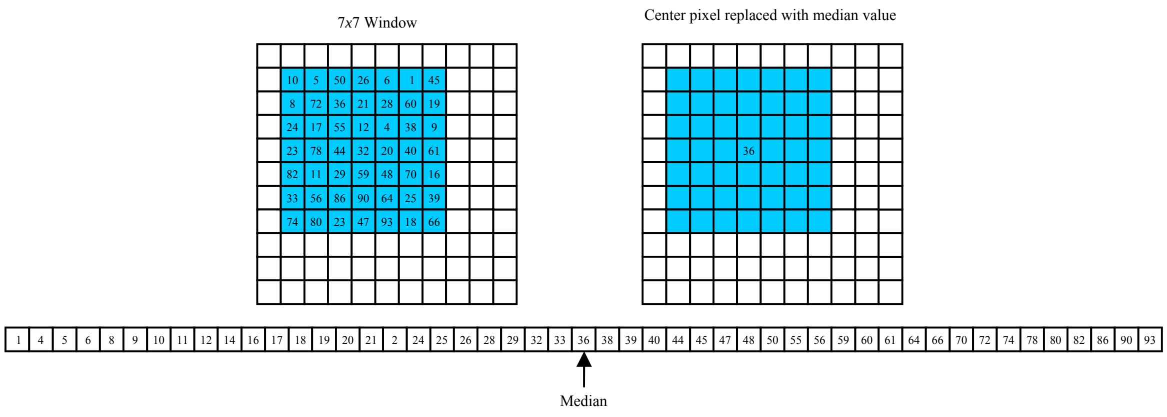


Figure 6.4 Concept of median filter

Median filtering implementation requires the operation with fixed-point numbers. For this application the architecture has been set to process a 640×480 input image and a 7×7 mask with fixed-point coefficients. In the same way that occurs with convolution, filtering requires that PEs perform multiplication followed by an accumulation. The internal structure of the architecture is not modified therefore the results obtained in Table 6.1 are kept. Figure 6.5 shows the output image processed by the FPGA architecture for a median filter.

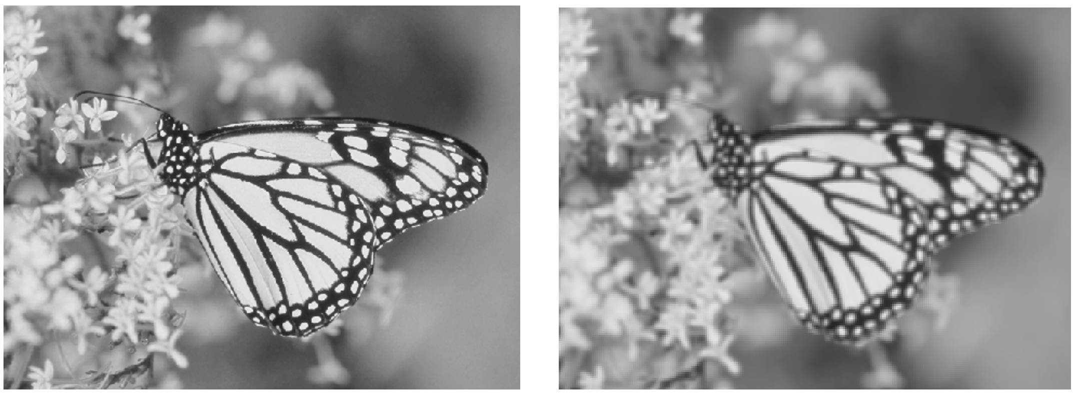

(a) Original image *(b) Filtered Image*

Figure 6.5 Output for median filter

6.2.2 High Pass Filtering

Edges are places in the image with strong intensity contrast. Edges often occur at image locations representing object boundaries; edge detection is extensively used in image segmentation the image is divided into areas corresponding to different objects.

Representing an image by its edges has the further advantage that the amount of data is reduced significantly while retaining most of the image information. Edges can be detected by applying a high pass frequency filter in the Fourier domain or by convolving the image with an appropriate kernel in the spatial domain [158]. In practice, edge detection is performed in the spatial domain, because it is computationally less expensive and often yields better results.

A high pass filter has been implemented in the systolic architecture using a window mask of 7×7. Figure 6.6 shows the resultant output image.

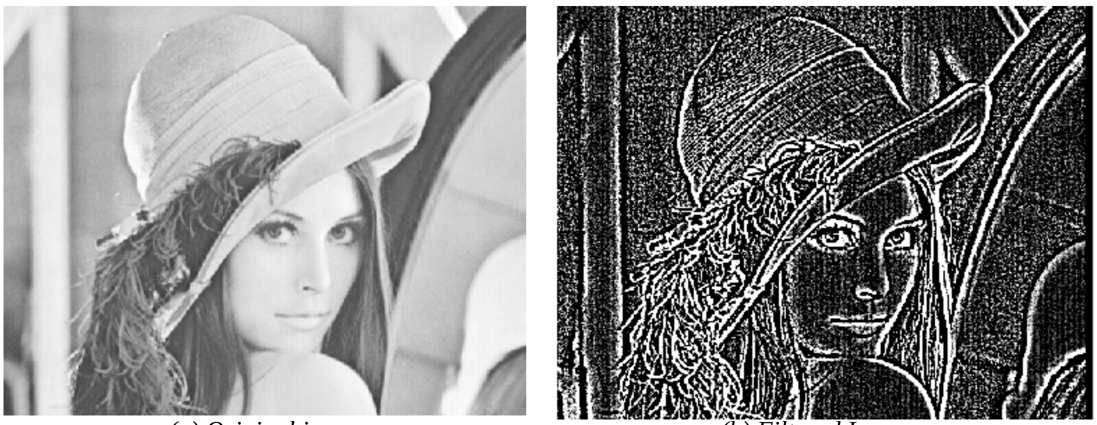

(a) Original image *(b) Filtered Image*

Figure 6.6 Output for high pass filter

6.3 Matrix Multiplication

Matrix multiplication is the fundamental operation in many numerical linear algebra applications. Its efficient implementation on parallel high performance system, together with the implementation of other basic linear algebra operations, is an issue of primary importance to be solved [159].

Given an M×N matrix A and an N×P matrix B, the result matrix C is given by Equation 6.1:

$$c_{i,j} = \sum_{r=1}^{n} a_{ir} b_{rj} \qquad (6.1)$$

For each pair i and j with $1 \le i \le M$ and $1 \le j \le P$.

This means that each value of C is a dot product of two vectors, one row from the A and one column from the B, which is illustrated in Figure 6.7.

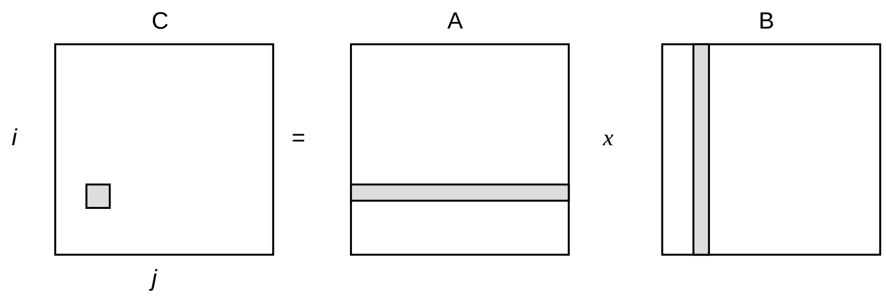


Figure 6.7 Matrix by matrix multiplication

In order to verify the correct functionality of the FPGA implementation the example presented in Figure 6.8 has been executed in the board.

$$A=\begin{bmatrix} 1 & 2 & 6 & 5 & 8 & 9 & 9 \\ 5 & 4 & 2 & 5 & 4 & 1 & 2 \\ 5 & 9 & 3 & 2 & 1 & 4 & 7 \\ 5 & 4 & 5 & 7 & 8 & 9 & 6 \\ 3 & 2 & 1 & 6 & 7 & 5 & 3 \\ 11 & 3 & 2 & 7 & 3 & 5 & 9 \\ 12 & 10 & 5 & 3 & 9 & 7 & 2 \end{bmatrix} \quad B=\begin{bmatrix} 7 & 2 & 8 & 5 & 1 & 19 & 9 \\ 5 & 41 & 2 & 5 & 9 & 10 & 2 \\ 7 & 22 & 3 & 2 & 6 & 4 & 8 \\ 5 & 4 & 5 & 17 & 8 & 19 & 6 \\ 3 & 12 & 5 & 6 & 14 & 5 & 1 \\ 1 & 3 & 2 & 27 & 13 & 5 & 9 \\ 2 & 9 & 5 & 30 & 9 & 17 & 2 \end{bmatrix}$$

$$AxB=\begin{bmatrix} 135 & 440 & 158 & 673 & 405 & 396 & 198 \\ 111 & 307 & 111 & 245 & 180 & 297 & 116 \\ 132 & 540 & 125 & 434 & 249 & 379 & 150 \\ 170 & 489 & 186 & 645 & 410 & 475 & 236 \\ 100 & 260 & 121 & 396 & 265 & 306 & 133 \\ 173 & 349 & 205 & 606 & 294 & 573 & 229 \\ 222 & 703 & 215 & 474 & 391 & 519 & 262 \end{bmatrix}$$

Figure 6.8 Matrix multiplication example

The architecture has been set to process two 7×7 matrixes. The PEs are configured to perform the scalar function of multiplication and the reduction function of accumulation over all coefficients in the matrixes. The technical data of post-place and route for the convolution architecture are shown in Table 6.2.

Table 6.2 Technical data for matrix multiplication

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	11,969 out of 19,200
Number 4 input LUTs	5,241 out of 38,400
Number of Flip Flops	1,628 out of 38,400
Number of Block RAMs:	13 out of 160
Overall % occupancy	62 %
Clock frequency	66 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	5.9 GOPs
Power Consumption	2.017 W

6.4 Pyramid Decomposition

A Gaussian pyramid is a set of images, where every image inside this set has a predecessor (with exception of the original image), and every image has ½ resolution with reference to its predecessor. To obtaining the images a Gaussian filter is used. Filter coefficients are obtained from a vector of weights [160]. The concept of pyramid is shown in Figure 6.9.

Consider an $M \times N$ image represented by the array G_0. Every pixel in the array represents a gray level for that point in the image. The image G_0 is considered as first level or Level 0 of the Gaussian pyramid. The second level of the pyramid contains the image G_1 which is a reduced version or filtered version of the image G_0. Every value inside the level 1 is obtained as a filtered value of the level 0's values using a Gaussian filter. The image of the level 2, represented for G_2, is obtained from the values of the level 1 image and the application of the same Gaussian filter.

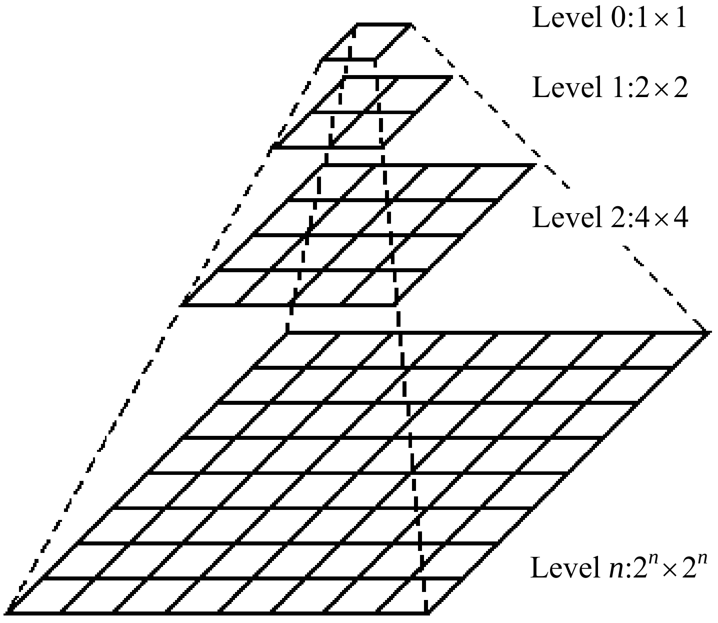


Figure 6.9 Gaussian Pyramid concept

The Gaussian mask used is the same for every level in the pyramid. This mask is obtained from a vector of weights with these characteristics:

a) The vector of weights is normalized

$$\sum_{m=-2}^{2}\hat{w}(m)=1 \tag{6.2}$$

b) The vector of weights is symmetrical

$$\hat{w}(i)=\hat{w}(-i) \tag{6.3}$$

For i = 0, 1, 2.

c) The mask must be detachable in relation to the vector of weights

$$w(m,n)=\hat{w}(m)\,\hat{w}(n) \tag{6.4}$$

The vector of weight is a discrete approximation to a Gauss function as shown in Figure 6.10. As the process to obtain a Gaussian pyramid, this vector of weights converges towards a continuous Gauss function, in such a way that in the infinite level of the pyramid, the vector of weights is equivalent to the continuous Gauss function.

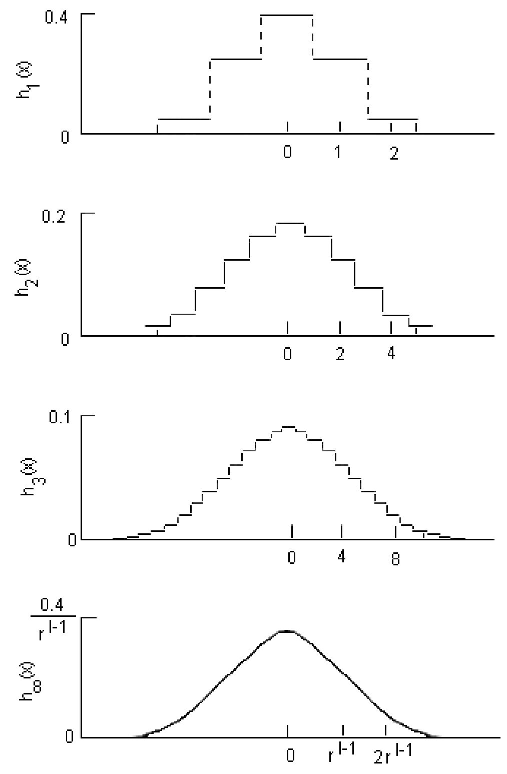


Figure 6.10 Gaussian approximation

For example, using the vector of weights from Table 6.3, which accomplished with the first two conditions aforementioned, the mask of Table 6.4 is obtained. The mask's values are obtained applying the third restriction

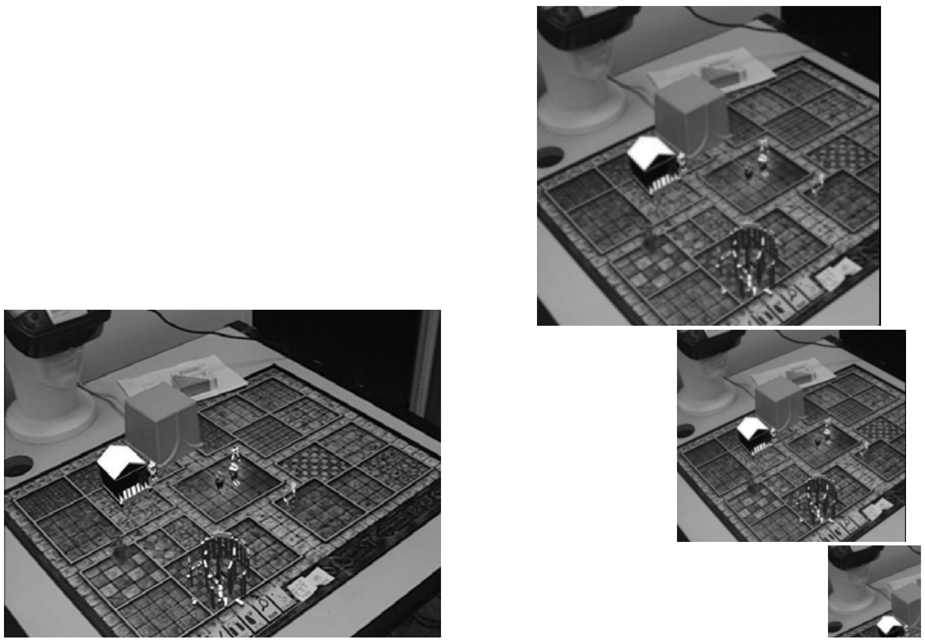
Table 6.3 Vector of weighs for the Gaussian filter

0.0500	0.1000	0.2000	0.3000	0.2000	0.1000	0.0500

Table 6.4 Mask generated from vector of weights

0.0025	0.0050	0.0100	0.0150	0.0100	0.0050	0.0025
0.0050	0.0100	0.0200	0.0300	0.0200	0.0100	0.0050
0.0100	0.0200	0.0400	0.0600	0.0400	0.0200	0.0100
0.0150	0.0300	0.0600	0.0900	0.0600	0.0300	0.0150
0.0100	0.0200	0.0400	0.0600	0.0400	0.0200	0.0100
0.0050	0.0100	0.0200	0.0300	0.0200	0.0100	0.0050
0.0025	0.0050	0.0100	0.0150	0.0100	0.0050	0.0025

To verify the architecture, a Gaussian filter of 7×7 has been used for the image shown in the Figure 6.11. In this case 2 levels for the pyramid have been chosen.



(a) Original image *(b) Processed Image*

Figure 6.11 Output image for 2 Level Gaussian Pyramid

The implementation of pyramid is based on the Gaussian filter implementation; therefore data in Table 6.1 corresponds with results obtained for this application.

6.5 Morphological Operators

The term morphological image processing refers to a class of algorithms that is interested in the geometric structure of an image [161]. Morphology can be used on binary and gray scale images, and is useful in many areas of image processing, such as skeletonization, edge detection, restoration and texture analysis [162].

A morphological operator uses a structuring element to process an image as shown in Figure 6.12. The structuring element is a window scanning over an image, which is similar to the mask window used in filters. The structuring element can be of any size, but 3×3 and 5×5 sizes are common. When the structuring element scans over an element in the image, either the structuring element fits or does not fit Figure 6.12 demonstrates the concept of a structuring element fitting and not fitting inside an image object.

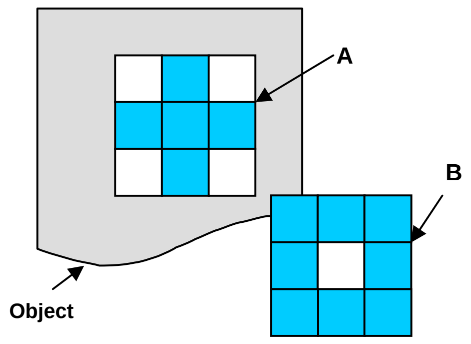


Figure 6.12 Concept of structuring element.
Element A fits in the object. Element B dos not fit

The most basic building blocks for many morphological operators are erosion and dilation [163]. Erosion as the name suggests is shrinking or eroding an object in an image. Dilation on the other hand increases the image object. Both of these objects depend on the structuring element and how it fits within the object.

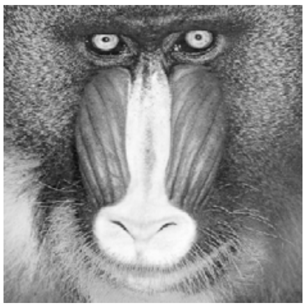
For example, if erosion is applied to a binary image, the resultant image is one image where there is a foreground pixel for every center pixel where its structuring element fit within an image. If dilation is applied, the output will be a foreground pixel for every point in the structuring element.

Important operations like opening and closing of an image can be derived by performing erosion and dilation in different order. If the erosion is followed by dilation, the resulting operation is called an opening. Closing operation is dilation followed by erosion. These two secondary morphological operations can be useful in image restoration, and their iterative use can yield further interesting results such as; skeletonization of an input image.

While morphological operations usually are performed on binary images, some processing techniques also apply to gray level images. These operations are for the most part limited to erosion and dilation. Gray level erosions and dilations produce results identical to the nonlinear minimum and maximum filters.

In a minimum filter, the center pixel in the moving window is replaced by the smallest pixel value. This has the effect of causing the bright areas of an image to shrink, or erode. Similarly, gray level dilation is performed by using the maximum operator to select the greatest value in a window.

When the gray-level erosion or dilation is mapped to the architecture, the scalar function corresponds to an addition/subtraction for erosion/dilation. The local reduction function corresponds to a maximum/minimum for erosion/dilation. The values of the structuring elements correspond to the window mask coefficients.
In order to test the correct operation of the erosion, the algorithm has been executed in the FPGA board. The resultant image obtained is observed in Figure 6.13 (b).


(a) Original image

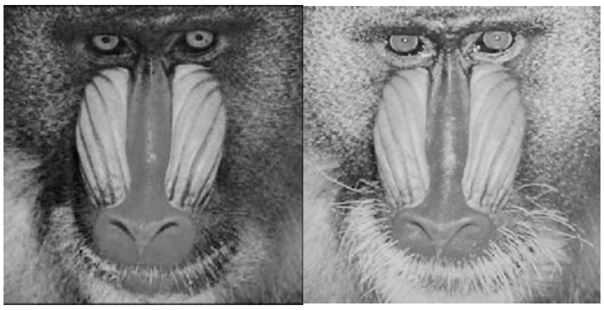

(b) erosion (c) dilation

Figure 6.13 Output for morphological operators

The architecture has been set to process a 640×480 input image and a 7×7 integer window mask. The processing elements are configured to perform the scalar function of subtraction and the reduction function of minimum over the image. The technical data of post-place and route for the erosion architecture are shown in Table 6.5.

To illustrate the dilation algorithm the PEs have been configured to perform the scalar function of addition and the reduction function of maximum over an input image of 640×480 using a 7×7 window mask. The resultant image is shown in Figure 6.13 (c). The corresponding post-place and route details for the dilation algorithm implementation are shown in Table 6.6.

Table 6.5 Technical data for erosion

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	12,114 out of 19200
Number 4 input LUTs	19,163 out of 38,400
Number of Flip Flops	4,613 out of 38,400
Number of Block RAMs:	13 out of 160
Overall % occupancy	63 %
Clock frequency	66 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	5.9 GOPs
Power Consumption	2.4 W

Table 6.6 Technical data for dilation

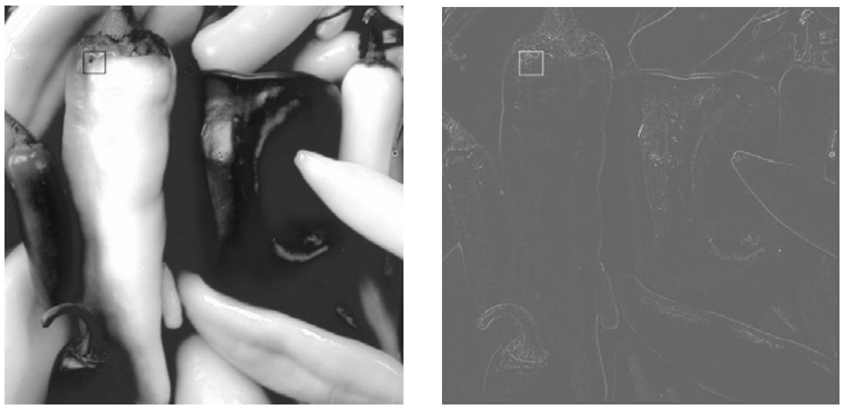
Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	12,074 out of 19200
Number 4 input LUTs	19,079 out of 38,400
Number of Flip Flops	4,571 out of 38,400
Number of Block RAMs:	13 out of 160
Overall % occupancy	62 %
Clock frequency	66 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	5.9 GOPs
Power Consumption	2.017 W

6.6 Template Matching

Template matching is one of the most fundamental tasks in many image processing applications. It is a simple method for locating specific objects within an image, where the template (which is, in fact, an image itself) contains the object that is being searched [164]. For each possible position in the image the template is compared with the actual image data in order to find subimages that match the template. To reduce the impact of possible noise and distortion in the image, a similarity or error measure is used to determine how well the template compares with the image data. A match occurs when the error measured is below a certain predefined threshold.

In template matching it is required a measure of dissimilarity between the intensity values of the template and the corresponding values of the image. Several measures may be defined for this purpose. One of the most popular is the SAD [165].

The template matching algorithm has been executed in the FPGA board considering a template of 7×7 that is searched over a 640×480 input image. In Figure 6.14 (a) the template used is shown in the original image and the resulting match is shown in Figure 6.14 (b).



(a) Original Image (b) Template Matching

Figure 6.14 Output for template matching operator

For this application the PEs are configured to perform the scalar function of subtraction and the reduction function of minimum. The technical data of post-place and route for the application are shown in Table 6.7.

Table 6.7 Technical data for template matching

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	12,114 out of 19200
Number 4 input LUTs	19,163 out of 38,400
Number of Flip Flops	4,613 out of 38,400
Number of Block RAMs:	13 out of 160
Overall % occupancy	63 %
Clock frequency	66 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	5.9 GOPs
Power Consumption	2.4 W

6.7 Performance Discussion

In this chapter some representative algorithms based on windows-operators convolution, filtering, matrix multiplication, pyramid decomposition, morphological operators and template matching have been presented in order to validate the correct functionality of the proposed architecture and its generalization as a hardware platform. The technical data presented for each version of the architecture constitute a measure of its performance. The three main parameters considered are the speed, the throughput and the power consumption. Table 6.8 summarizes the results obtained for this set of algorithms.

Table 6.8 Summary of the architecture performance

Application	Number of Slices	Clock Frequency	Power Consumption
Convolution	11,969 out of 19200	66 MHz	2.017 W
Filtering	11,969 out of 19200	66 MHz	2.017 W
Matrix multiplication	11,969 out of 19200	66 MHz	2.017 W
Gaussian pyramid	11,969 out of 19200	66 MHz	2.017 W
Erosion	12,114 out of 19200	66 MHz	2.4 W
Dilation	12,074 out of 19200	66 MHz	2.017 W
Template matching	12,114 out of 19200	66 MHz	2.4 W

From this table it can be observed little variations in the area occupied according to the algorithm being performed. These changes are due to the configuration selected for the PE's and the scalar operation being performed. However the performance and power consumption practically remain the same.

In order to establish the advantages of the presented architecture, the results obtained in Table 6.8 needs to be compared with previous implementations of image processing architectures; even though most performance metrics are rarely reported for architectures and systems in literature. This lack of standard metrics for comparison makes difficult to determine the advantages of a given system.

A. Dehon [13] proposed a model to compute the hardware resource utilization in a system considering the fabrication technology. This model provides a standard metric that allows doing a fair comparison between systems measuring the silicon area in feature size units rather than in absolute units.

The silicon area required by the architecture is computed in terms of the feature size λ. Considering data for XCV2000E device and the results obtained by A. Dehon and C. Torres [141] the chip area required for a CLB of an FPGA on a 180 nm process is given by the Equation 6.6:

$$A_{CLB} = 35462 \mu m^2 = 4.4x10^6 \lambda^2 \qquad (6.5)$$

Where λ is the feature size and it is equal to 90 nm.

Based on equation 6.5 the total amount of silicon area for the complete architecture can be computed with equation 6.6.

$$A_{Total} = \# CLBs \ x \ A_{CLB} \qquad (6.6)$$

For each version of the architecture presented, the area occupied is computed based on the number of slices reported by the synthesis tool. The results are shown in Table 6.9.

Table 6.9 Amount of area for applications of the architecture

Application	**Number of CLBs**	**Silicon Area**	
Convolution	5985	212.24×10^6 μm2	26.4 Gλ^2
Filtering	5985	212.24×10^6 μm2	26.4 Gλ^2
Matrix multiplication	5985	212.24×10^6 μm2	26.4 Gλ^2
Gaussian pyramid	5985	212.24×10^6 μm2	26.4 Gλ^2
Erosion	6057	214.79×10^6 μm2	26.7 Gλ^2
Dilation	6037	214.08×10^6 μm2	26.6 Gλ^2
Template matching	6057	214.79×10^6 μm2	26.7 Gλ^2

Considering all the results presented in this chapter it is possible to present a comparison with previous architectures. For this purpose the execution time, given in milliseconds, and the silicon area occupied are considered as main metrics. The assessments were made considering that the systems deal with the same algorithm and they use the same image size. Table 6.10 present the technical details for the chosen architectures.

Table 6.10 Performance for different architectures

System	Architecture	Application	Image Size	Timing	Silicon Area
R. Lopez [152]	SIMD FPGA-based	3×3 Filtering	640×480	23.04 ms	Not reported
M. Vega [153]	FPGA-based	3×3 Filtering	640×480	868.51 ms	322 $G\lambda^2$
C. Torres [141]	Systolic FPGA-based	7×7 Generic Window-based Image operator	640×480	9.7 ms	15 $G\lambda^2$
M. Vega [166]	Systolic FPGA-based	7×7 Median Filter	640×480	998.20 ms	1.41 $G\lambda^2$
Pentium III, 1GHz. [167]	Von Newman	3×3 Generic Convolution	640×480	2863 ms	N/A
Proposed Architecture	Systolic	7×7 Generic Window-based operators	640×480	5 ms	26.7 $G\lambda^2$

In summary, the proposed architecture provides a throughput of 5.9 GOPs on a chip area of 26.7 $G\lambda^2$ with an estimated power consumption of 2.017 W running at 66 MHz clock frequency. From these results it can be shown that it is possible to achieve real-time performance for applications based on windows operators. Furthermore, the capacity of generalization for the proposed schema has been established.

This point is resumed again in next chapter for motion estimation which is considered independently to this set of algorithms due to its higher complexity. Showing that the architecture is capable of giving support to such algorithms the potential of the proposal can be demonstrated.

Chapter 7

Motion Estimation

Motion Estimation (ME) is a basic bandwidth compression method used in video-coding systems that requires a huge amount of computation; this fact justifies the great research effort that has been made to develop efficient dedicated architectures and specialized processors for motion estimation [168-171].

The main objective of the present architecture is to give support to algorithms that present a high computational complexity and represent a challenge for its implementation for real time performance, they must follow a regular pattern in data movements and they must use reduced and efficient memory space.

In order to verify the presented architecture potential, Motion Estimation has been also implemented, considering a new memory schema for the Full Search Block Matching Algorithm (FSBMA) [172]. The present design keeps the total memory size and the number of transfers as small as possible, while maintaining high throughput that is required for motion estimation applications. The implementation of this algorithm proves the generalization of the architecture.

7.1 Motion Estimation Algorithm

Motion estimation is a key technique in most algorithms for video compression such as Moving Picture Coding Experts Group (MPEG) [173] and H.26L that exploits the spatial and temporal redundancies present in a digital video sequence [174, 175]. Nevertheless, it is also the most computationally intensive task, involving up to 65% of the total computational resources of the video coder and requiring high power, throughput and memory utilization [176]. All these factors are critical design metrics for most novel video applications, aimed at portable and battery supplied terminal devices using limited bandwidth communication channels [177].

In order to reduce the computational complexity of motion estimation algorithms many methods have been proposed, such as block matching algorithms, parametric/motion models, optical flow, and pel-recursive techniques [176]. Among these approaches, the Full Search Block Matching algorithm is the most common due to its effectiveness and simplicity for both software and hardware implementations [178].

To implement motion estimation in coding image applications, the most popular and widely used method, due to its easy implementation, is the FSBMA. The FSBMA divides the image in squared blocks (macro-block) and compares each block in the current frame (reference block) with those within a reduced area of the previous frame (search area) looking for the best match [179]. The matching position relative to the original position is described by a motion vector, as shown in Figure 7.1.

$I_k(x, y)$ is defined as the pixel intensity at location (x, y) in the k-th frame and $I_{k-1}(x, y)$ is the pixel intensity at location (x, y) at the k-1-th frame. For FSBMA motion estimation, $I_{k-1}(x, y)$, represents usually a pel located in the search area of the size $R^2 = R_x \times R_y$ pel of the reference frame and $I_k(x, y)$ belongs to the current frame. The block size is defined as $N^2 = N \times N$ pel. Each individual search position of a search schema is defined by $\overrightarrow{CMV} = (dx, dy)$.

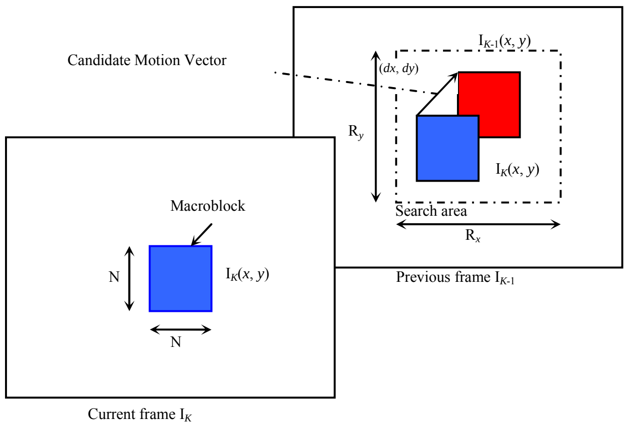


Figure 7.1 Block-matching for motion estimation.

In block matching the SAD [180] is usually adopted as matching criteria to determine the displacement between macroblocks under processing and a group of pixels defined within a search region in a previous frame.

The matching procedure is made by determining the optimum of the selected cost function, usually SAD, between the blocks. The SAD is defined as:

$$SAD(dx,dy)=\sum_{m=x}^{x+N-1}\sum_{n=y}^{y+N-1}\left|I_k(m,n)-I_{k-1}(m+dx,n+dy)\right| \tag{7.1}$$

$$\overrightarrow{MV}=\left(MV_x,MV_y\right)=\min_{(dx,dy)\in R^2} SAD(dx,dy) \tag{7.2}$$

The motion vector $\overrightarrow{MV}$ represents the displacement of the best block with the best result for the distance criterion, after the search procedure is finished.

Previously, there has been several hardware architectures reported in the literature to cope with real time and high volume requirements of motion estimation algorithms [181-183]. These architectures make use of massive pipelining and parallel processing provided by systolic [184-186] or linear arrays [187]. Most of them require two separate memories for storing the current frame and the previous frame increasing their size; hence, efficient memory utilization becomes one of the most important design problems. Furthermore, they are not intrinsically power efficient [141].

7.2 Implementation

In order to implement the ME algorithm, three features must be taken into account. First, the FSBMA provides the most accurate results but is computationally expensive. Second, SAD is usually adopted due to its simpler computational complexity and satisfactory results. Finally, the large amount of data managed, mainly in the search area, demands highly efficient data-flow.

As it can be observed the motion estimation demands correspond with window-based operators. Therefore the hardware architecture needs some special characteristics that provide support for both algorithms. The FSBMA must be represented as a window operator in order to fit in the architecture though some adjustments are needed.

Due to the nature of Equation 7.1 the FSBMA can be formulated as a window-based operator considering the following aspects:

- The coefficients of the window mask are variable and new windows are extracted from the first image to constitute the reference block. Once the processing in the search area has been completed, the window mask must be replaced with a new one, and the processing goes on the same way until all data is processed.

- The different windows to be correlated are extracted in a column-based order from the search area to exploit data overlapping and sharing. The pixels are broadcasted to all the processors to work concurrently.

Based on these characteristics, the processing block has been modified to support not only the SAD operations required for FSBMA, but the rest window-operations involved in low level image processing.

A Macro-Block (MB) corresponds to the whole array, with every PE representing a pixel from the block. At every clock cycle a column of pixels from the reference and the current image is broadcasted to all the PEs in a column to calculate the SAD and shift the value to a partial results collector in charge of accumulate results located in the same column of the array and the captured results are sent to the global data collector. The GDC stores the result of a MB processed and sends it to the output memory buffer.

The SAD value for each column is compared to a reference or the minimum of previously calculated SAD values, with the lower value being retained till the end of the search area is reached. From that moment on, successive Macro-Blocks are processed in the horizontal direction reusing data stored in the input buffers. Reading image pixels from buffers one row below, it is possible to traverse the image in the vertical direction. Data for processing is available in a row format therefore when blocks are processed vertically; some data in the search area are overlapped for two blocks as shown in Figure 7.2.

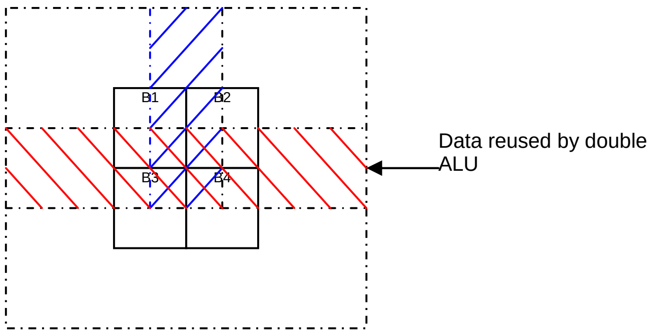


Figure 7.2 Data overlapped between search areas

in the horizontal and vertical direction.

At the beginning of the application the system can be configured to define the number of ALU's to be presented during computation. For the case of Low-level processing, simple ALUs are implemented in the architecture, but if the selected operation is Motion Estimation, the double ALU schema is active.

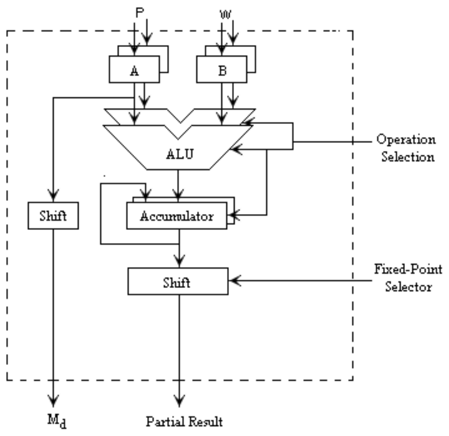


Figure 7.3 Processing element for motion estimation implementation

The result obtained from post place and route process for the motion estimation version of the architecture are shown in Table 7.1

Table 7.1 Technical data for the entire architecture

Element	Specification
Virtex-E	XCV2000E
FPGA technology	0.18 μm 6-layer metal process
Number of PEs	49
Number of Slices	12,100 out of 19200
Number 4 input LUTs	5,600 out of 38,400
Number of Flip Flops	7,742 out of 38,400
Number of Block RAMs:	18 out of 160
Overall % occupancy	63%
Clock frequency	60 MHz
Off-chip memory data buses	21 bit-address, 32 bit data
Internal data buses for ALUs	8 bits for fixed-point operations
Peak performance	~9 GOPs
Power Consumption	3 W

For a couple of images from a video sequence the resulting post synthesis motion vectors are shown in Figure 7.4.

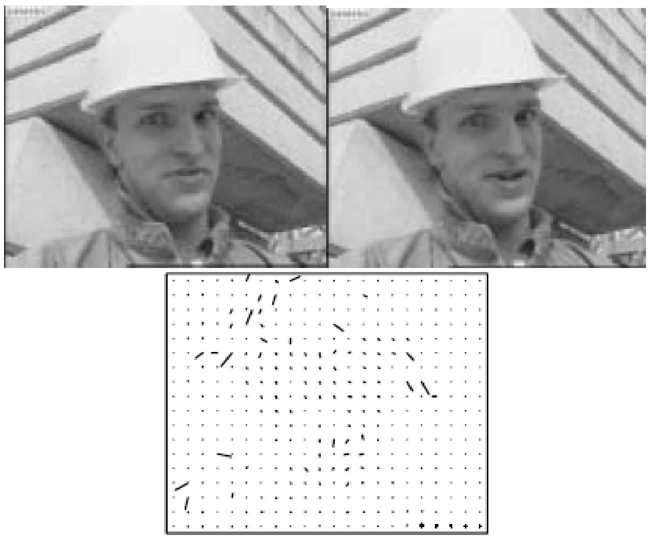


Figure 7.4 Motion vectors for image sequence

7.3 Discussion

With the implementation of FSBMA the flexibility and robustness of the proposed architecture has been demonstrated. The memory and data flow schemas used have been used successfully achieving a high throughput of $\simeq$9 GOPs with a clock frequency of 60 MHz. The obtained results show the possibility of video rate performance. In comparison with the frequency achieved in the window-operators architecture, it has been a reduction; this is due to the growth in resources and the router needs associated.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

In this thesis an architecture dedicated to window operations in image processing using a 2D systolic array has been presented. The architecture implemented is flexible enough to support several variations of window-based image operators.

Input data are stored in small buffers that allow regular access and reuse of data. This reduces the need for internal data storage. The schemas proposed for data flow and memory management contribute to achieve high parallel efficiency and low area-time product.

The high level pipeline model proposed allows the operation scheduling and process chaining which conducts to provide a solution to a broader number of image processing algorithms.

The FPGA-based architecture allows the designer to implement a systolic array with a variable size, as well as optimize the buffer memory size used for a particular problem according to a chosen sequence of operators.

The obtained results show that the area utilization is very compact and the power consumption is reduced making the architecture suitable for embedded systems. The use of FPGA technology has proved to be a promising alternative to implement efficiently novel image processing architectures under real-time.

The architecture processing capacity can be improved if aspects of dynamic reconfiguration are explored.

8.2 Discussion

As image processing applications consist of several phases with different computer architectural needs, hardware reconfiguration is an important provision. With the implementation of the proposed architecture it has been shown that reconfiguration can be employed within the low-level processing stage achieving high performance. Analysis of the results obtained shows several benefits:

- First, the image processing application have been accelerated; the architecture produces an output result on each clock cycle after a latency period, proportional to the size of the window used, and performs seven arithmetic operations concurrently for a window mask of 7×7. The latency arises at the beginning of processing since the columns of the systolic array must be full in order to output a result. The architecture provides a throughput of 5.9 GOPs which implies performance in real time.
- Second, using a reconfigurable system for image processing can save significant amount of hardware. The original target application takes advantage of the reconfiguration to set up different scripts for the main process. The performance of the hardware based system coupled with a group of image buffers opens the possibility of a very complex multi-function image processor with essentially the same per-unit hardware cost as a single function system.

- Third, the use of image buffers benefits the application as the number of accesses to the input image memory is reduced. The buffers take data from the input memory following a circular pipeline schema. The buffer content is refreshed during the systolic array operation, hiding in this way the memory accessing time. These storage elements give the opportunity of data reuse because once the data are placed in the buffers; they can be rerouted to different elements to be processed. Inside the buffers the 2D parallelism becomes more obvious with the storage of the image pixels as neighboring elements.
- The global bus inserted in the architecture outlines the routing facilities to speed up the transfer of both control and configuration parameters during program execution and I/O memory transfers. This particular element extends the capabilities of the systolic array giving more flexibility.

With the addition of all these features, the proposed architecture performance is comparable with other image processing architectures presented in the literature.

8.3 Future Work

The work presented here is a step in the direction of automatic mapping of algorithms to a set of hardware libraries for near-optimal execution.

There are, however, a number of issues that could be addressed as the next possible steps for improving this process:

- To maximize data re-use and to minimize reconfiguration time are critical to an optimal execution schedule.
- To extend the architecture capabilities to a broader subset of algorithms.

- Automatically map the algorithms or a part of them to the architecture in order to accelerate some video applications.

Even though the area occupation of the architecture is not very large, it is possible to improve the hardware resource utilization via dynamic reconfiguration techniques. Partial reconfiguration allows the possibility of loading different designs into the same area of the FPGA device or the flexibility to change portions of the design without having either reset or completely reconfigure the entire device.

References

[1] Kung, H. T. and Leiserson, C. E. "*Systolic Arrays for VLSI*", Technical Report CS 79-103, Carnegie Mellon University, 1978.

[2] Bouldin, D., "*Enabling Killer Applications of Reconfigurable Systems*", Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, June 2005, pp.7-16.

[3] Ramachandran, S. and Srinivasan, S., "*FPGA Implementation of a Novel, Fast Motion Estimation Algorithm for Real-Time Video Compression*", Proceedings of the International Symposium on Field Programmable Gate Arrays, February 2001, pp. 213-219.

[4] Benkrid, K., et al., "*High Level Programming for FPGA based Image and Video Processing using Hardware Skeletons*", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2001, pp. 219-226.

[5] Siegel, H. J., Armtrong, J. B. and Watson, D. W., "*Mapping Computer Vision-Related Tasks onto Reconfigurable Parallel-Processing System*", IEEE Computer, Vol.25, No. 2, February 1992, pp. 54-63.

[6] Ye, A. G. and Lewis, D. M., "*Procedural Texture Mapping on FPGAs*", Proceedings of the 1999 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, February 1999, pp. 112-120.

[7] Compton, K., "*Reconfigurable Computing: A survey of System and Software*", ACM Computing Surveys, Vol. 34, No.2, June 2002, pp. 171-210.

[8] Jacomet, M., et al., "*On a development environment for real-time information processingin system-on-chip solutions*", Proceedings of the 14th Symposium on Integrated Circuits and Systems Design, Pirenopolis, Brazil, September 2001, pp. 28-31.

[9] Gribbon, K. T., Bailey, D. G. and Johnston, C. T., "*Design Patterns for Image Processing Algorithm Development on FPGAs*", Proceedings of the IEEE TENCON 2005, Melbourne, Australia, November 2005, pp. 1-6.

[10] Bois, B., Bois, G. and Savaria, Y. "*Reconfigurable pipelined 2-D convolvers for fast digital signal processing*", IEEE Transactions on VLSI, Vol. 7, No. 3, 1999, pp. 299-308.

[11] Managuli, R., et al., "*Mapping of two dimensional convolution on very long instruction word media processors for real-time performance*", Journal of Electronic Imaging 2000, Vol. 9, Issue 3, pp. 327-335.

[12] Kim, J., Cho, J. and Kim, T. G., "*Temporal Partitioning to Amortize Reconfiguration Overhead for Dynamically Reconfigurable Architectures*", IEICE Transactions on Information and Systems, Vol. E90-D, No. 12, June 2007, pp. 1977-1985.

[13] DeHon, A., "*The Density Advantage of Configurable Computing*," IEEE Computer, Vol. 33, No. 4, April 2000, pp. 41-49.

[14] El-Ghazawi, T., "*Is High-Performance Reconfigurable Computing the Next Supercomputing Paradigm*?", Proceedings of the ACM/IEEE Supercomputing Conference, November 2006, pp. XV-XV.

[15] Awalt, R. K. "*Making the ASIC/FPGA Decision*", Integrated System Design Magazine, July 1999, pp. 22-28.

[16] Ghiasi, S., Nahapetian, A. and Sarrafzadeh, M., "*An Optimal Algorithm for Minimizing Run-Time Reconfiguration Delay*", Transactions on Embedded Computing Systems, Vol. 3, No. 2, May 2004, pp. 237–256.

[17] Xilinx, Home Page, `URL: http://www.xilinx.com`

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

[19] Ratha, N. K. and Jain, A. K., "*Computer Vision Algorithms on Reconfigurable Logic Arrays*", IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 1, 1999, pp. 29-43.

[20] Xilinx. Virtex™ 2.5 V Field Programmable Gate Arrays (DS003-2), ver. 2.8.1, December 2002. Available at `http://www.xilinx.com/`.

[21] Hauck, S., "*The Roles of FPGAs in Reprogrammable Systems*", Proceedings of the IEEE, Vol. 86, No. 4, April 1998, pp. 615-639.

[22] Chang, K. C., "*Digital design and modeling with VHDL and synthesis*", IEEE Computer Society Press, Washington, 1997, 345 p.

[23] Thomas, D. E., "*The Verilog Hardware Description Language*", Kluwer Academic Publishers, Boston 1996, 310 p.

[24] Celoxica Limited, "*Handel-C Language Reference Manual*", 2005.

[25] Poznanovic, D., "*Application Defined Processors*", Linux Journal, December 2004, `http://www.linuxjournal.com/article/7731`

[26] Stitt, G., et al., "*Using On-Chip Configurable Logic to Reduce Embedded System Software Energy*", 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, 2002, pp. 143-151.

[27] DeHon A. and Wawrzynek, J., "*Reconfigurable Computing: What, Why, and Implications for Design Automation*", Proceedings of 36th Design Automation Conference, New Orleans, Louisiana, 1999, pp. 610-615.

[28] Schoner, B., Jones, C. and Villasenor, J., "*Issues in Wireless Coding Using Run-Time-Reconfigurable FPGAs*," Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, April 1995, pp. 85-89.

[29] Eldredge, J. G. and Hutchings, B. L., "*Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration*", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, April 1994, pp.180-188.

[30] Eldredge, J.G. and Hutchings, B. L., "*Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs*", Journal of VLSI Signal Processing", Vol. 12, 1996, pp. 67-86.

[31] Antoni, L., Leveugle, R., and Feher, B., "*Using run-time reconfiguration for fault injection in hardware prototypes*", Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, October 2000, pp. 405-413.

[32] Rincon, F. and Teres, L., "*Reconfigurable Hardware Systems*", International Semiconductor Conference, Vol.1, October 1998, pp.45-54.

[33] Mcmillan, S. and Guccione, S., "*Partial Run-Time Reconfiguration Using JRTR*", Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications, August 2000, pp. 352 – 360.

[34] MacBeth, J. and Lysaght, P., "*Dynamically Reconfigurable Cores*", Lecture Notes in Computer Science, August 2001, 462 p.

[35] Enzler, R., Plessl, C. and Platzner, M., "*System-level performance evaluation of reconfigurable processors*", Microprocessors and Microsystems, Vol. 29 No. 2–3, April 2005, pp.63–73

[36] Shohei, A., et al., "*An Adaptive Viterbi Decoder on the Dynamically Reconfigurable Processor*", Proceedings of the International Conference on Field Programmable Technology (FPT'06), December 2006, pp. 285-288.

[37] Enzler, R., Plessl, C. and Platzner, M., "*Virtualizing Hardware with Multi-Context Reconfigurable Arrays*", Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03), September 2003, pp. 151-160.

[38] Robinson, D. and Lysaght, P., "*Verification of Dynamically Reconfigurable Logic*", Proceedings of the Field Programmable Logic and Applications international congress (FPL'00), Villach, Austria, August 2000, pp. 141-150.

[39] JHDL, Home Page, `URL: http://www.jhdl.org`

[40] Poetter, A. et al., "*JHDLBits: The Merging of Two Worlds*", Proceedings Field-Programmable Logic and Applications international congress (FPL'04), Leuven, Belgium, August 2004, pp. 414-423.

[41] Guccione, S., Levi, D. and Sundararajan, P., "*JBits: Java Based Interface for Reconfigurable Computing*", Proceedings of the Military and Aerospace Applications of Programmable Devices and Technologies International Conference, 1999, `URL:http://citeseer.ist.psu.edu/guccione99jbits.html`

[42] Celoxica, Home Page, `URL: http://www.celoxica.com`

[43] Culler, D., Singh J. P. and Gupta, A., "*Parallel Computer Architecture: A Hardware/Software Approach*", Morgan Kaufmann Publishers, 1998, 1100 p.

[44] Sutherland, I. E., "*Micropipelines*", Communications of the ACM Vol. 32, No. 6, June 1989, pp. 720 - 738.

[45] Ikenaga T, Ogura T., "*A Fully Parallel 1-Mb CAM LSI for Real-Time Pixel-Parallel Image Processing*", IEEE Journal of Solid-State Circuits, Vol. 35, No. 4, April 2000, pp. 536-544.

[46] Serratosa, F., Millán, P. and Montseny, E., “*Systolic processors applied to computer vision systems*”, Proceedings of the Computer Architectures for Machine Perception (CAMP’95), 1995, pp. 178-183.

[47] Kung, H. T., “*Why systolic architectures?*”, IEEE Computer, Vol. 15, No. 1, 1982, pp. 37-46.

[48] Dong, K., et al., “*Research on Architectures for High Performance Image Processing*”, Proceedings of The International Workshop on Advanced Parallel Processing Technologies, September 2001, pp. CDROM.

[49] Athanas, P. M. and Abbott, A. L., “*Real-time image processing on a custom computing platform*”, IEEE Computer Vol. 28, No. 2, 1995, pp. 16-24.

[50] DK Design Suite.
`http://www.celoxica.com/products/dk/default.asp`

[51] Fatemi, H., et al., “*Run-time reconfiguration of communication in SIMD architectures*”, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, April 2006, pp. 213-216.

[52] Maresca, M. and Lavin, M. A., “*Parallel Architectures for Vision*”, Proceedings of the IEEE, Vol. 76, No. 8, 1988, pp. 970-981.

[53] Hammerstrom, D. and Lulich, D., “*Image Processing Using One Dimensional Processor Array*”, Proceedings of the IEEE, Vol. 84, No. 7, 1996, pp. 1005-1018.

[54] Crisman, J. D. and Webb, J. A., "*The Warp Machine on Navlab*", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 13, No. 5, 1991, pp. 451-465.

[55] Duff, M. J. B., "*CLIP 4: A large scale integrated circuit array parallel processor*", Proceeding IEEE Intl. Joint Conf. On Pattern Recognition, 1976, pp. 728-733.

[56] Batcher, K., "*Design of a massively parallel processor*", IEEE Transactions on Computers, Vol. 29, No. 9, 1980, pp. 836-840.

[57] Tucker, L. W. and Robertson, G. G., "*Architecture and Applications of the Connection Machine*", IEEE Computer, Vol. 21, No. 8, 1988, pp. 26-38.

[58] Cantoni, V., et al., "*Papia: Pyramid architecture for parallel image analysis*", Proceedings of the 7th IEEE symposium on Computer Arithmetic, June 1985, pp. 237-242.

[59] Merigot, A., Zavidovique, B. and Devos, F., "*SPHINX. A pyramidal approach to parallel image processing*", Proceedings of the IEEE Workshop Computer Architecture for Pattern Analysis and Image Database Management, 1985, pp. 107-111.

[60] Lee, W., et al., "*Space-time scheduling of instruction-level parallelism on a Raw machine*", International Conference on Architectural Support for Programming Languages and Operating Systems, 1998, pp. 46-57.

[61] Goldstein, S. C., et al., "*PipeRench: a coprocessor for streaming multimedia acceleration*", International Symposium on Computer Architecture, Atlanta, GA, 1999, pp. 28-39.

[62] Rixner, S., et al., "*A bandwidth-efficient architecture for media processing*", Proceedings of the ACM/IEEE 31st international symposium on Microarchitecture, December 1998, pp. 3-13.

[63] Hsieh, C. and Kim, S. P., "*A Highly-Modular Pipelined VLSI Architecture for 2-D FIR Digital Filter*", Proceedings of the IEEE 39th Midwest Symposium on Circuits and Systems, August 1996, pp. 137-140.

[64] Haule, D. D. and Malowany, A. S., "*High-speed 2-D Hardware Convolution Architecture Based on VLSI Systolic Arrays*", Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, June 1989, pp. 52-55.

[65] Hsu, K., et al., "*A Pipelined ASIC for Color Matrixing and Convolution*", Proceedings of the IEEE 3rd ASIC Seminar and Exhibit, September 1990, pp. P7/6.1-P7/6.6.

[66] Hecht, V., Rönner K. and Pirsch, P., "*An Advanced Programmable 2D-Convolution Chip for Real Time Image Processing*", Proceedings of IEEE International Symposium on Circuits and Systems, June 1991, pp. 1897-1900.

[67] Chang, H. M. and Sunwoo, M. H., "*An Efficient Programmable 2-D Convolver Chip*", Proceedings of the IEEE International Symposium on Circuits and Systems, June 1998, pp. 429-432.

[68] Annaratone, M., et al., "*The Warp Computer: Architecture, implementation and Performance*", IEEE Transactions on Computers, Vol. 36, No. 12, 1987, pp. 1523-1538.

[69] Borkar, S., et al., "*Supporting Systolic and memory communication in iWarp*", Proceedings of the 17th annual international symposium on Computer Architecture, 1990, pp. 70-81.

[70] Chai, S. M. and Wills, D. S., "*Systolic Opportunities for Multidimensional Data Streams*", IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 4, 2002, pp. 388-398.

[71] Siegel, H. J., et al., "*PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition*", IEEE Transactions in Computers C-30, 1981, pp. 934-946.

[72] Weems, C. C., et al., "*The image understanding architecture*", International Journal of Computer Vision, Vol. 2, No. 3, 1989, pp. 251-282.

[73] Hayes J. P., et al., "*Architecture of a hypercube supercomputer*", Proceedings of 1986 Int. Conf. on Parallel Processing, 1986, pp. 653-660.

[74] The INTEL concurrent computer. INTEL Corp, Portland, OR, 1985.

[75] Max Video product literature, Datacube Corp., Peabody, MA, 1987.

[76] Rieger, C., Bane, J. and Trigg, R.., "*A highly parallel multiprocessor*", Proceedings of IEEE Workshop on Picture Data Description and Management, 1980, pp. 298-304.

[77] Withby-Strevens, C., "*The transputer*", Proceedings of 72th ACM Int. Symp. on Computer Architectures, 1985, pp. 292-300.

[78] Sternberg, S. R.., "*Biomedical image processing*", IEEE Computer, Vol. 16, No.1, January 1983, pp. 22-34.

[79] Kent, E. W., Shneier, M. 0. and Lumia, R., "*PIPE: Pipeline image processing engine*", Parallel and Distributed Computing. Vol. 2, 1987, pp. 50-78.

[80] Batcher, K. E., "*Bit-serial parallel processing systems*", IEEE Transactions on Computers, Vol. C-31, 1982, pp. 377-384.

[81] Oldfield, D. E. and Reddaway, S. F., "*An image understanding performance study on the ICL distributed array processor*", IEEE Comp. Sci. Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Miami Beach, FL, 1985, pp. 256-264.

[82] Kondo, T., et al., "*An LSI adaptive array processor. In IEEE Solid-state Circuits*", Vol. SC-18, 1983, pp. 147-156.

[83] Robinson, I. N. and Moore, W. R., "*A parallel processor array architecture and its implementation in Silicon*", Proceedings of IEEE Custom Integrated Circuit Conf., 1982, pp. 41-45.

[84] Shaw, D. E., "*The NON-VON supercomputer*", Technical Report, Computer Science Department, Columbia University, New York, NY, 1982.

[85] Nudd, G. R., et al, "*The application of three-dimensional microelectronics to image analysis*", Integrated Technology for Parallel Image Processing, S. Levialdi, Ed. New York, NY: Academic Press, 1986, 236 p.

[86] Bouknight, W. L., et al, "*The ILLIAC IV system*", Proceedings of IEEE, Vol. 60, No. 4, April 1972, pp. 369-388.

[87] Fountain, T. J., "*Array architectures for iconic and symbolic image processing*". Proceedings of the 8th Int. Conf. on Pattern Recognition, October 1986, pp. 24-33.

[88] Beetem, J., Denneau, M. and Weingarten, D. H., "*The GFll supercomputer*", Proceedings of 12th Int. Symp. on Computer Architectures, 1985, pp. 108-118.

[89] Li, H. and Maresca, M., "*Polymorphic-torus network*", Proceedings of Int. Conf. on Parallel Processing, 1987, pp. 411-414.

[90] Yun, H. K. and Silverman, H. F., "*A Distributed Memory MIMD Multi-Computer with Reconfigurable Custom Computing Capabilities*", International Conference on Parallel and Distributed Systems, Vol. 10, No. 13, December 1997, pp. 8-13.

[91] Ratha, N. K., Jain, A. K. and Rover, D. T., "*Convolution on splash 2*", 3rd. IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'95), 1995, pp. 204-213.

[92] Ebeling, C., et al., "*Mapping applications to the RaPiD configurable architecture*", 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97), 1997, pp. 106-115.

[93] Wiatr, K. and Russek, P., "*Embedded Zero Wavelet Coefficient Coding Method for FPGA Implementation of Video Codec in Real-Time Systems*", The International Conference on Information Technology: Coding and Computing (ITCC'00), 2000, pp. 146-151.

[94] Perry, S., et al., "*SIMD 2-D Convolver for Fast FPGA-based Image and Video Processors*", Proceedings of the MAPLD International Conference, September 2003. pp. CDROM-D2.

[95] Bajwa, R. S., Owens, R. M. and Irwin, M. J., "*Mixed-Autonomy Local Interconnect for Reconfigurable SIMD Arrays*", IEEE Fourth International Conference on High-Performance Computing, 1997. pp. 428-431.

[96] Cloutier, J., et al., "*VIP: An FPGA-based Processor for Image Processing and Neural Networks*", IEEE Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro'96), 1996. pp. 330-336.

[97] Kung, H. T. and Leiserson, C. E., "*Algorithms for VLSI processor arrays*", Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980, Chapter 8.

[98] Myers, G. J., Yu, Y. C. and House, D. L., "*Microprocessor Technology Trends*", IEEE Proceedings of Special Issue on IC Technologies of the Future, 1986, pp. 1605-1622.

[99] Kung, H. T. "*Systolic algorithms for the CMU WARP processor*", Proceedings of the Seventh Int. Conf. on Pattern Recognition, July 1984, pp. 570-577.

[100] Petkov, N. "*Systolic Parallel Processing*", Elsevier Science Inc., New York, USA, 1992, 712 p.

[101] Hillis, D. and Barnes, J. "*Programming a Highly Parallel Computer*", Nature No. 326, 1987, pp. 27-30.

[102] Perri, S., et al., "*A high-performance fully reconfigurable FPGA-based 2D convolution processor*", Microprocessors and Microsystems Vol. 29, 2005, pp 381-391.

[103] Ratha, N. K., Jain, A. K. and Rover, D. T., "*FPGA-based coprocessor for text string extraction*", 5th. IEEE International Workshop on Computer Architecture for Machine Perception, 2000, pp. 217-221.

[104] Ratha, N. K. and Jain, A. K., "*FPGA-based computing in computer vision*", 4th. IEEE International Workshop on Computer Architecture for Machine Perception, 1997, pp. 128-137.

[105] Dudek, P. and Carey, S. J., "*A General-Purpose 128$\times$128 SIMD Processor Array with Integrated Image Sensor*", Electronics Letters, vol.42, no.12, June 2006, pp.678-679.

[106] Rosas, R. L., De Luca, A. and Santillan, F. B. "*SIMD architecture for image segmentation using Sobel operators implemented in FPGA technology*", Proceedings of the 2nd International Conference on Electrical and Electronics Engineering, September 2005, pp. 77-80.

[107] Mozef, E., et al., "*Parallel Architecture Dedicated to Connected Component Analysis*", Proceedings of the International Conference on Pattern Recognition (ICPR'96), August 1996, pp. 699-703.

[108] Seinstra, F. and Koelma, D., "*Modeling Performance of Low Level Image Processing Routines on MIMD Computers*", Proceedings of the Fifth Annual Conference of the Advanced School for Computing and Imaging (ASCI'99), June 1999, pp. 307-314.

[109] Jamro, E. and Wiatr, K., "*Implementation of Convolution Operation on General Purpose Processors*", Proceedings of the 27th Euromicro Conference, September 2001, pp. 410-417.

[110] Marrtins, S. and Alves, J. C., "*A high-level tool for the design of custom image processing systems*", Proceedings of the 8th Euromicro Conference on Digital System Design (DSD'05), 2005, pp. 346-349.

[111] Fatemi, H., et al., "*Implementing Face Recognition Using a Parallel Image Processing Environment Based on Algorithmic Skeletons*", Proceedings of the Annual Conference of the Advanced School for Computing and Imaging (ASCI'04), June 2004, pp. 351-357.

[112] Nakagawa, A. et al., "*Combining Words and Object-Based Visual Features in Image Retrieval*", Proceedings of the 12th International Conference on Image Analysis and Processing (ICIAP'03), September 2003, pp. 354-359.

[113] Barta, A. and Vajk, I., "*Integrating Low and High Level Object Recognition Steps by Probabilistic Networks*", International Journal of Information Technology, Vol. 3, No. 1, 2006, pp. 64-73.

[114] Kim, S. and Kweon, I. S., "*Multi-modal Sequential Monte Carlo for On-Line Hierarchical Graph Structure Estimation in Model-based Scene Interpretation*", Proceedings of the 18th International Conference on Pattern Recognition (ICPR'06), 2006, pp. 251-254.

[115] Le Beux, S., Marquet, P. and Dekeyser, J. L., "*Multiple Abstraction Views of FPGA to Map Parallel Applications*", Proceedings of the Reconfigurable Communication-centric SoCs (ReCoSoC'07), Montpellier, France, June 2007.

[116] Hockney, R. W. and Jesshope, C. R., "*Parallel Computer-2: architecture, programming and environments*", 2nd edition, Inst. of Phys./Adam Hilger, Bristol, 1992, 625 p.

[117] Hord, R. H., "*Parallel Supercomputer in SIMD Architectures*", CRC Press. Inc., 1990, 400 p.

[118] Plimptom, S., Mastin, G. and Ghiglia, D., "*Synthetic aperture radar image processing on parallel supercomputers*", Proceedings of the ACM/IEEE conference on Supercomputing, 1991, pp. 446-452.

[119] Skillicorn, D. B., "*Foundations of Parallel Programming*", Cambridge Series in Parallel Computation 6, Cambridge University Press, 2005, 209 p.

[120] Hwang, K., "*Advanced Computer Architectures: Parallelism, Scalability, Programmability*", McGraw-Hill, Higher Education, 1992, 672 p.

[121] Veale, V. F., Antonio, J. K. and Tull, M. P., "*Configuration Steering for a Reconfigurable Superscalar Processor*", Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3, April 2005, pp. 152b-159b.

[122] Shen, J. P. and Lipasti, M. H., "*Modern Processor Design: Fundamentals of Superscalar Processors*", McGraw-Hill, New York, July 2004, 642 p.

[123] Talla, D., et al., "*TMS320DM310 – A portable digital media processor*", Proceedings of IEEE HOT Chips, Stanford University, August 2003.

[124] Talla, D., "*Architectural techniques to accelerate multimedia applications on general-purpose processors*", Department of Electrical and Computer Engineering, The University of Texas, Austin Texas, August 2001.

[125] James-Roxby, P., Schumacher, P. and Ross, C., "*A Single Program Multiple Data Parallel Processing Platform for FPGAs*", Proceedings of the 12th Symposium on Field-Programmable Custom Computing Machines (FCCM'04), April 2004, pp. 302-303.

[126] McBader, S. and Lee P., "*An FPGA Implementation of a Flexible, Parallel Image Processing Architecture Suitable for Embedded Vision Systems*", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), April 2003, pp. 228a-232a.

[127] Reed, D. and Hoare, R., "*An SoC Solution for Massive Parallel Processing*", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02), April 2002, pp. 245-252.

[128] Inoguchi, Y., "*Outline of the Ultra Fine Grained Parallel Processing by FPGA*", Proceedings of the Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region (HPCAsia'04), July 2004, pp. 434-441.

[129] Baumstark, L. B. and Wills, L. M., "*Retargeting Sequential Image-Processing Programs for Data Parallel Execution*", IEEE Transactions on Software Engineering, Vol. 31, No. 2, February 2005, pp. 116-136.

[130] Flynn, M. J., "Some Computer Organizations and *Their Effectiveness*", IEEE Transactions on Computers, September 1972, pp. 948-960.

[131] Duncan, R., "*A Survey of Parallel Computer Architectures*", IEEE Computer, Vol. 23, No. 2, February 1990, pp. 5-16.

[132] Johnson, K. T. and Hurson, A. R., "*General-Purpose Systolic Arrays*", IEEE Computer, Nol. 26, No. 11, November 1993, pp. 20-31.

[133] Jordan H. F., "*Fundamentals of Parallel Processing*", Prentice Hall, 2003, 536 p.

[134] Song, S. W., "*Systolic algorithms: concepts, synthesis, and evolution*", CIMPA School of Parallel Computing. Temuco, Chile, 1994, 41 p.

[135] Rao, D. V., et al., "*Implementation and Evaluation of Image Processing Algorithms on Reconfigurable Architecture using C-based Hardware Descriptive Languages* ", International Journal of Theoretical and Applied Computer Sciences, Vol. 1, No. 1, 2006, pp. 9-34.

[136] Umbaugh, S. E., "*Computer Vision and Image Processing-a practical approach using CVIP tools*", Prentice Hall, 1997, 528 p.

[137] Li, D., Jiang, L. and Kunieda, H., "*Design optimization of VLSI array processor architecture for window image processing*", IEICE Transactions on Fundamentals, Vol. E82-A, No. 8, 1999, pp. 1474-1484.

[138] Li, D., Jiang, L. and Kunieda, H., "*Design Optimization of VLSI Array Processor Architecture for Window Image Processing*", IEICE Trans. Fundamentals, Vol. E82-A, No. 8, 1999, pp. 1474-1484.

[139] Kasturi R. and Shunck, B. G. "*Machine vision*", McGraw-Hill, New York, 1995, 549 p.

[140] Koenderink, J. J. and Van Doorn, A. J., "*Generic Neighborhood Operators*", In IEEE Transactions on pattern analysis and machine intelligence, Vol. 14, No. 6, 1992, pp. 597-605.

[141] Torres-Huitzil C., "*Reconfigurable Computer Vision System for Real-time Applications"*, Ph.D. Thesis, INAOE, Mexico, 2003.

[142] Managuli, R., et al., "*Mapping of two-dimensional convolution on very long instruction word media processors for real-time performance*", Journal of Electronic Imaging, Vol. 9, No. 3, 2000, pp. 327-335.

[143] Steven, J. E., Wilton, S. A. and Wayne, L., "*The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays*", Proceedings of the Field-Programmable Logic and Applications international congress (FPL'04), 2004, pp. 719-728.

[144] Torres-Huitzil, C. and Arias-Estrada, M. "*Real-time image processing with a compact FPGA-based systolic architecture*", Real Time Imaging, No.10, 2004, pp. 177-187.

[145] Hughey, R., "*Programming Systolic Arrays*", Proc. Int. Conference on Application-Specific Array Processors, Berkeley, CA, USA, August 1992, pp. 604-618.

[146] Devos, H., et al., "*Performance Requirements for Reconfigurable Hardware for a Scalable Wavelet Video Decoder*", Proceedings of ProRISC, 2003, pp. 56-63

[147] Rutenbar, R. A., "(*When*) *will FPGAs kill ASICs*?", Proceedings of the 38th conference on Design automation, 2001, pp. 321 - 322

[148] Lacassagne, L., Etiemble D. and Ould Kablia, S. A. "*16-bit floating point instructions for embedded multimedia applications*", Proceedings of the Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05), July 2005, pp. 198-203.

[149] Tang, T. Y., Moon, Y. S. and Chan K. C., "*Efficient Implementation of Fingerprint Verification for Mobile Embedded Systems using Fixed-point Arithmetic*", Proceedings of the Symposium on Applied Computing, 2004, pp.821-825.

[150] Cilio A., Karkowski, I. and Corporaal, H. "*Fixed-point arithmetic for ASIP code generation*", Proceedings of the 4th conference of the Advanced School for Computing and Imaging (ASCI'98), June 1998, pp. 44-50.

[151] Girod, B., "*What's wrong with mean-squared error*", Digital images and human vision, MIT Press, Cambridge, MA, USA, 1993, pp. 207-220.

[152] Rosas, R. L., De Luca, A. and Santillan, F. B., "*SIMD architecture for image segmentation using Sobel operators implemented in FPGA technology*", 2nd International Conference on Electrical and Electronics Engineering (ICEEE), September 2005, pp. 77-80.

[153] Vega-Rodriguez, M. A., Sanchez-Perez, J. M. and Gomez-Pulido, J. A., "*An optimized architecture for implementing image convolution with reconfigurable hardware*", Proceedings of the World Automation Congress, Vol. 16, June-July 2004, pp. 131-136.

[154] Awcock, G. J. and Thomas, R., "*Applied Image Processing*", McGraw-Hill, USA, 1996, 300 p.

[155] Haralick, R. M. and Shapiro, L. G., "*Computer and Robot Vision*", Prentice-Hall, USA, 2002, 630 p.

[156] Chan, R. H., Ho, C. W. and Nikolova, M., "*Salt-and-Pepper Noise Removal by Median-Type Noise Detectors and Detail-Preserving Regularization*", IEEE Transactions on Image Processing, Vol. 14, No. 10, October 2005, pp. 1479-1485.

[157] Pitas, I. and Venetsanopoulos, A. N., *"Edge detectors based on nonlinear filters"*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 4, July 1986, pp. 538-550.

[158] Nibouche, M., et al., "*Rapid Prototyping of Orthonormal Discrete Wavelet Transforms on FPGAs*" IEEE International Symposium on Circuit and Systems, Sydney, Australia, Vol. 3, May 2001, pp. 1399-1402.

[159] Dou Y., et al., "*64-bit Floating-Point FPGA Matrix Multiplication*", Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays FPGA'05, Monterrey, California, USA, February 2005, pp. 86-95.

[160] Burt, P. J. and Adelson H. E., "*The Laplacian Pyramid as a Compact Image code*", IEEE Transactions on Communications, Vol. 31, No. 4, April 1983, pp. 532-540.

[161] Sinha, D. and Giardina C. R., "*Discrete Black and White Object Recognition via Morphological Functions*", Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, No. 3, March 1990, pp. 275-293.

[162] Dlamantaras I. and Kung S Y, "*A Linear Systolic Array for Real-Time Morphological Image Processing*", Journal of VLSI Signal Processing, Vol. 17, No. 1, 1997, pp. 43-55.

[163] Muthukumar, V. and Rao, D. V., "*Image processing algorithms on reconfigurable architecture using Handel-C*", Proceedings of the EUROMICRO Symposium on Digital System Design (DSD'04), September 2004, pp. 218-223.

[164] Hezel, S., et al., "*FPGA-based Template Matching using Distance Transforms*", Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Washington, DC, USA, 2002, pp. 89-97.

[165] Tombari, F., Di Stefano, L. and Mattoccia, S., "*A robust measure for visual correspondence*", Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP'2007), 2007, pp. 376-381.

[166] Vega-Rodriguez, M. A., Sanchez-Perez, J. M. and Gomez-Pulido, J. A., "*An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems*", Proceedings of the 10th Mediterranean Conference on Control and Automation (MED'2002), Lisbon, Portugal, July 2002, pp. 131-136.

[167] Herrmann, C. and Langhammer, T., “*Automatic Staging for Image Processing*”, Technical Report, Fakultät für Mathematik und Informatik, Universität Passau, Germany 2004.

[168] Oktem, S. and Hamzaoglu, I. “*A Quarter Pel Full Search Block Motion Estimation Architecture for H.264/AVC*”, Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007), August 2007, pp. 444-447.

[169] Zandonai, D., Bampi, S. and Bergerman, M. “*ME64 — A Highly Scalable Hardware Parallel Architecture Motion Estimation in FPGA*”, Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI’03), September 2003, pp. 93-98.

[170] McErlean, M., “*An FPGA Implementation of Hierarchical Motion Estimation for Embedded Object Tracking*”, Proceedings of the IEEE International Symposium on Signal Processing and Information Technology, August 2006, pp. 242-247.

[171] Ryszko, A. and Wiatr, K., “*An Assesment of FPGA Suitability for Implementation of Real-Time Motion Estimation*”, Proceedings of the Euromicro Symposium on Digital Systems Design (DSD’01), September 2001, pp. 364-367.

[172] Roma, N. and Sousa, L., “*A New VLSI Architecture for Full Search Block Matching*”, Proceedings of the Eleventh International Conference on Very Large Scale Integration of Systems-on/Chip: SOC Design Methodologies, 2001, pp. 253 – 264.

[173] Kuhn, P., et al., "*Complexity and PSNR-Comparison of Several Fast Motion Estimation Algorithms for MPEG-4*", Proc. Applications of Digital Image Processing (SPIE), Vol. 3460, San Diego, July 1998, pp. 486-499.

[174] Kuhn, P., "*Algorithms, complexity analysis and VLSI architectures for MPEG-4 Motion Estimation*", Kluwer Academic Publisher, USA, 1999.

[175] Chandrakasan, A., Sheng, S. and Brodersen, R., "*Low-power CMOS digital design*", IEEE J. Solid-State Circuits, Vol. 27, No. 4, 1992, pp. 473–484.

[176] Dias, T., Roma, N. and Sousa, L., "*Fully Parameterizable VLSI Architecture for Sub-Pixel Motion Estimation with Low Memory Bandwidth Requirements*", Terceiras Jornadas de Engenharia de Electrónica e Telecomunicações e de Computadores (JETC'05), Nov. 2005, pp. CDROM.

[177] Kuhn, P., et al., "*Complexity and PSNR-Comparison of Several Fast Motion Estimation Algorithms for MPEG-4*", Proc. Applications of Digital Image Processing (SPIE), San Diego, Vol. 3460, July 1998, pp. 486-499.

[178] Dufaux, F. and Moscheni, F., "*Motion estimation techniques for digital TV: a review and a new contribution*", Proc. IEEE, Vol. 83, No. 6, June 1995, pp. 858-879.

[179] Yang, K., et al., "*A family of VLSI designs for the motion compensation block-matching algorithm*", IEEE Trans. on Circuits and Systems, Vol.36, Oct. 1989, pp.1317-1325.

[180] Gui-guang, D. and Bao-long, G., "*Motion Vector Estimation Using Line-Square Search BlockMatching Algorithm for Video Sequences*", EURASIP Journal on Applied Signal Processing 2004, Vol.11, pp. 1750-1756.

[181] Vassiliadis, S., et al., "*The sum-absolute-difference motion estimation accelerator*," 24th. EUROMICRO Conference, Vol. 2, August 1998, pp. 20559-20566.

[182] Komarek, T. and Pitsch, P., "*Array architectures for blockmatching algorithms*", Trans. On Circuits and Systems, Vol.36, No.10, Oct. 1989, pp. 1301-1308.

[183] Pirsch, P., Demassieux, N. and Gehrke, W., "*VLSI architectures for video compression*", Proceedings of the IEEE, Vol.83, No.2, February 1995, pp. 220-246.

[184] Sung, M., "*Algorithms and VLSI architectures for motion estimation*", VLSI Implementations for Image Communications, P. Pirsch (Ed.), 1993, pp. 251-2281.

[185] Hsieh, C. and Lin, T., "*VLSI Architecture for block-matching motion estimation algorithm*", IEEE Trans. on Circuits and Systems for Video Technology, Vol.2, No.2, June 1992, pp.169-175.

[186] Chan, E., et al., "*Motion estimation architecture for video compression*", IEEE Trans. Consumer Electronics, Vol.39, No.3, August 1993, pp. 292-297.

[187] Baek, J., et al., "*A fast array architecture for block matching algorithm*", Proc. of IEEE Symposium on Circuits and Systems (ISCAS), Vol. 4, June 1994, pp. 211-214.

Appendix A. Glossary

Block RAM (BRAM). Units of RAM embedded in Xilinx Virtex and Spartan FPGAs. Each Block RAM is dual ported, and can be configured in a range of widths and depths.

Configurable Logic Block (CLB). The basic tile type in Xilinx Virtex FPGAs. A CLB comprises four slices, two tristate buffers (in the Virtex-II and Virtex-II Pro families) and routing, connected to the general routing resources by a switch matrix.

EDIF. Electrical Design Interchange Format. Standard format for representing electronic designs.

Look up table (LUT). An n-LUT is a look up table with n inputs and one output, capable of implementing any combinational logic function of n inputs.

Netlist. List of logic gates and interconnections comprising a circuit, such as an EDIF file.

Programmable Logic Device (PLD). A generic name for semiconductor devices which can be programmed or configured post-fabrication to implement a variety of circuits.

Processing Element. A modular circuit block which implements a certain processing task.

Slice. The basic configurable logic unit within a configurable logic block. Each slice comprises two 4-LUTs, two registers as well as multiplexer logic and other specialised circuitry such as fast carry chains.

Switch matrix. A configurable interconnect resource inside each reconfigurable tile of Virtex FPGAs, the switch matrix connects the logic and routing within the tile to general routing resources of the FPGA.

Very Large Scale Integration (VLSI). A device with many tens of thousands of logic gates.

Appendix A. Architecture Data Sheet

Systolic array

Main features

- High speed configurable systolic array for low-level image processing
- Capacity for processes chaining

Functional description

This core support operations involved in common low-level image processing algorithms. It is based on a 2D customizable systolic array of processing elements that can be configured according to a control word. The core can carry out image filtering, morphological operations, matrix-matrix multiplication, template matching and pyramid processing. A streaming router takes data from/to input/output image memories and makes explicit the data parallelism usually found in the image processing. The incoming data is stored in internal memory buffers before being processed in parallel. An internal control bus is in charge of interchange parameters to customize the operation performed by core. The produced data by a processing module can be captured by an output data router and then transmitted to an external memory output.

Requirements

- External memories

Interface

A block diagram of the module interface with the signal names for the inputs and outputs is shown in figure 1.

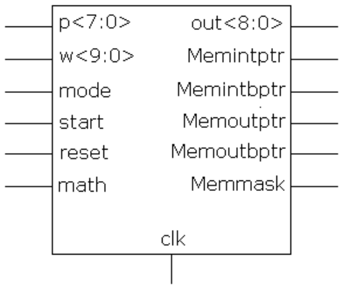


Figure 1. Input-output for the core

Table 1 summarizes the signal names and a short description of its functionality is provided.

Signal	Direction	Description
P	In	Pixels of the input image
W	In	Coefficients of the window mask
Mode	In	Selector for application: Window-based algorithm/Motion Estimation
Start	In	Start of the image processing
Reset	In	Reset signal for the architecture
Math	In	Data type selector: Integer/Fixed-point
Clk	In	Main clock
Out	Out	Pixels of the input image
Meminptr	In	Pointer to external input memory
Meminbptr	In	Pointer to input buffer memories
Memoutptr	In	Pointer to the external output memory
Memoutbptr	In	Pointer to output buffer memories
Memmask	In	Pointer to mask coefficients bank

Table 1. Core signal description

Core resource utilization

The area resource utilization of the core when targeted to a specific FPGA device is summarized in table 3. The results were obtained with the standard optimization effort of the Xilinx ISE tools.

	Configuration I
Xilinx part	Virtex-E
Area (Slices)	12114
FPGA percentage	63%

Table 2. Hardware resource utilization

Performance characteristics

The values in table 3 show the clock speed that can be achieved with other performance parameters. Different results can be obtained using different options for the pace and route stages in the FPGA implementation or using devices with faster speeds.

	Configuration I
Xilinx part	Virtex E
Maximum clock frequency	66 MHz
Data per second	61440000
Images per second	200

Table 3. Performance characteristics

Status

- Handel-C source code
- Debugged and validated through test bench
- Board tested

Motion Estimation

Main features

- Real time motion estimation on VGA sized gray level images
- Full search block matching

Functional description

This core implements the Full Search Block Matching Algorithm which can be represented as a window-based algorithm. It is based on a 2D customizable systolic array of processing elements that includes a double ALU in order to search multiple macro-blocks in parallel. A streaming router takes data from/to input/output image memories and makes explicit the data parallelism usually found in the image processing. The incoming data is stored in internal memory buffers before being processed in parallel. An internal control bus is in charge of interchange parameters to customize the operation performed by core. The produced data by a processing module can be captured by an output data router and then transmitted to an external memory output.

Requirements

- External memories

Interface

A block diagram of the module interface with the signal names for the inputs and outputs is shown in figure 1.

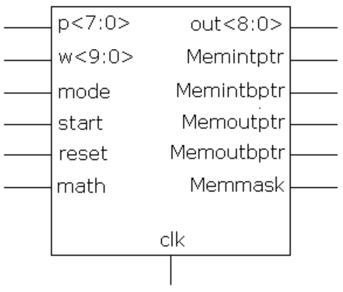


Figure 1. Input-output for the core

Table 1 summarizes the signal names and a short description of its functionality is provided.

Signal	Direction	Description
P	In	Pixels of the input image
W	In	Coefficients of the window mask
Mode	In	Selector for application: Window-based algorithm/Motion Estimation
Start	In	Start of the image processing
Reset	In	Reset signal for the architecture
Math	In	Data type selector: Integer/Fixed-point
Clk	In	Main clock
Out	Out	Pixels of the input image
Meminptr	In	Pointer to external input memory
Meminbptr	In	Pointer to input buffer memories
Memoutptr	In	Pointer to the external output memory
Memoutbptr	In	Pointer to output buffer memories
Memmask	In	Pointer to mask coefficients bank

Table 1. Core signal description

Core resource utilization

The area resource utilization of the core when targeted to a specific FPGA device is summarized in table 3. The results were obtained with the standard optimization effort of the Xilinx ISE tools.

	Configuration I
Xilinx part	Virtex-E
Area (Slices)	14847
FPGA percentage	77%

Table 2. Hardware resource utilization

Performance characteristics

The values in table 3 show the clock speed that can be achieved with other performance parameters. Different results can be obtained using different options for the pace and route stages in the FPGA implementation or using devices with faster speeds.

	Configuration I
Xilinx part	Virtex E
Maximum clock frequency	66 MHz
Data per second	50995200
Images per second	166

Table 3. Performance characteristics

Status

- Handel-C source code
- Debugged and validated through test bench
- Board tested
- Require extensions to be embedded

Appendix B. Academic Report

As a result of this thesis work, the following papers have been published in different outstanding forums.

Saldaña G., Arias-Estrada M., "*Compact FPGA-based systolic array architecture suitable for vision systems*", 4th International Conference on Information Technology: New Generations (ITNG'07), Las Vegas, Nevada, April, 2007.

Saldaña G., Arias-Estrada M., "*Compact FPGA-based systolic array architecture for motion estimation using full search block matching*", Southern Conference on Programmable Logic 2007 (SPL'07), Mar de Plata, Argentina, February, 2007.

Saldaña G., Arias-Estrada M., "*Customizable FPGA-Based Architecture for Video Applications In Real Time*", 2006 IEEE International Conference on Field Programmable Technology (FPT'06), Bangkok, Thailand, December, 2006.

Saldaña G., Arias-Estrada M., "*Real Time FPGA-based Architecture for Video Applications*", International Conference on Reconfigurable Computing and FPGAs, RECONFIG'06, San Luis Potosi, Mexico, September 2006.

Saldaña G., Arias-Estrada M., "*FPGA-Based Customizable Systolic Architecture for Image Processing Application*"s, International Conference on Reconfigurable Computing and FPGAs (RECONFIG'05), Puebla, México, September 2005.

Saldaña G., Arias-Estrada M., "*Real-time Computer Vision using FPGA based Processing: Overview of INAOE Activities*", Retine Electronique, Asic-FPGA et

DSP pour la Vision et le Traitement D'images en Temps Reel (READ’05), Evry, France, June 2005.

Saldaña G., Arias-Estrada M., “*FPGA systolic-based Architecture for Video Applications in Real Time*”, 7º Encuentro Internacional de Ciencias de la Computación 2006 (ENC’06), San Luís Potosí, Mexico, September 2006.

Saldaña G., Arias-Estrada M., “*Systolic-Based Architecture Suitable For Motion Estimation*”, 7to. Encuentro de Investigación. Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, Mexico, November 2006.

Saldaña G., Arias-Estrada M., “*Customizable Systolic Architecture for Image Processing Applications*”, 6to. Encuentro de Investigación. Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, Mexico, November 2005.

Saldaña G., Arias-Estrada M., “*Arreglo Sistólico Reconfigurable para el Procesamiento a Bajo Nivel de Imágenes”*, 5to. Encuentro de Investigación. Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, Mexico, November 2004.

Saldaña G., Arias-Estrada M., “*Arquitectura SIMD Reconfigurable*”, 4to. Encuentro de Investigación. Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, Mexico, November 2003.