Implementing a Highly Parameterized Digital PIV System On Reconfigurable Hardware

A Thesis Presented

by

Abderrahmane Bennis

The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
COMPUTER ENGINEERING

Advisor:

Dr. Miriam Leeser

Dissertation Committee:

Dr. Gilead Tadmor
and
Dr. Russ Tedrake

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

NORTHEASTERN UNIVERSITY

Boston, Massachusetts
2010

NORTHEASTERN UNIVERSITY
Graduate School of Engineering

Thesis Title: Implementing a Highly Parameterized Digital PIV System On Reconfigurable Hardware

Author: Abderrahmane Bennis.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Doctor of Philosophy Degree

---

Thesis Advisor: Prof. Miriam Leeser Date

Thesis Committee Member: Prof. Gilead Tadmor Date

Thesis Committee Member: Prof. Russ Tedrake Date

Department Chair: Prof. Ali Abur Date

Graduate School Notified of Acceptance:

Director of the Graduate School: Prof. Yaman Yener Date

NORTHEASTERN UNIVERSITY
Graduate School of Engineering

Thesis Title: Implementing a Highly Parameterized Digital PIV System On Reconfigurable Hardware

Author: Abderrahmane Bennis.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Doctor of Philosophy Degree

---

Thesis Advisor: Prof. Miriam Leeser Date

Thesis Committee Member: Prof. Gilead Tadmor Date

Thesis Committee Member: Prof. Russ Tedrake Date

Department Chair: Prof. Ali Abur Date

Graduate School Notified of Acceptance:

Director of the Graduate School: Prof. Yaman Yener Date

Copy Deposited in Library:

Reference Librarian Date

Abstract

Parameterization of circuits is increasingly in demand. It opens the door both for investigating the right parameters for different application domains and reuse of components with the specific parameters in building new custom hardware. This can significantly reduce the time to market. In this work, we investigate the parameterization of particle image velocimetry (PIV), a technique that is used in many engineering domains.

This thesis presents PARPIV, the design and prototyping of a highly parameterized digital PIV system implemented on reconfigurable hardware. Despite many improvements to PIV methods over the last twenty years, PIV post-processing remains a computationally intensive task. It becomes a serious bottleneck as camera acquisition rates reach 1000 frames per second. In addition, for different engineering applications, different PIV parameters are required. Up to now, there has been no PIV system that combines both flexible parameterization and high computational performance. In our research we have created such a system. This implementation is highly parameterized, supporting adaptation to a variety of setups and application domains. The circuit is parameterized by the dimensions of the captured images as well as the dimensions of the

interrogation windows and sub-areas, pixel representation, board memory width, displacement and overlap. Through this work, a parameterized library of different VHDL components was built. To the best of the author's knowledge, this is the first highly parameterized PIV system implemented on reconfigurable hardware reported in the literature. We report on the speedup in hardware over a standard software implementation of different implementations with different parameters. The parameterized PIV circuit has been designed and implmented on an ADM-XRC-5LX FPGA board from Alpha-Data. The computational speed of the hardware presents an average of 50x speedup in comparison with a software implementation running on a 3 GHz PC.

Acknowledgements

First of all, I would like to thank my advisor Professor Miriam Leeser for her precious support and invaluable guidance during my graduate research and study. Without her belief in me and her encouragement, this thesis wouldn't exist, and for this reason, I would like to express my sincere gratitude to her. Professor Leeser is to me a mentor, advisor, and an excellent researcher model to follow. I consider myself very lucky to have her as my PhD advisor, she is not only a great professional doctor, she is a very nice person with accurate perception and refined manners. I am humbled and honored to be one of her graduate students.

I would like to thank as well Prof. Tadmor and Prof. Tedrake who serve in my defense committee for their invaluable advices and supportive comments.

I would like to thank all my colleagues and friends for their help and support since my first day at the Reconfigurable Computing Laboratory, I would like to thank Wang Chen, Haiqian Yu, Xiaojun wang, Ben Cordes, Al Conti, Nick Moore, and Sherman Braganza. I would like also to thank Mary Ellen Fuess for her cooperation with the Alpha-data board.

I would like to express my special thanks to my parents, my brothers, and my family for their endless encouragement and great support. I would like to dedicate this work for them.

This work was sponsored by the NSF, the National Science Foundation and the CenSSIS, the Center of Subsurface Sensing and Imaging Systems.

Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

1.1 Overview

Field Programmable Gate Arrays (FPGA) provide a platform for accelerating the running time of applications, especially digital signal processing applications where massive data are processed and high performance computing is demanded. The flexibility and the affordability of FPGAs make them very competitive in the semiconductor industry so they find applications across the entire spectrum of electronic system design. The FPGAs market size reached $1.9 billion in 2005 and it was expected to increase to $2.75 billion in 2010. However, revenues of 2007 exceeded the expectations and achieved $3.6 billion by 2007 and the market is still expanding [1-2]. Of that Xilinx has 50%, Altera has 36%, and Actel and Lattice each have about 7%. Typical FPGA applications belong to the

fields of networking equipment, communication devices, medical devices, consumer electronics, defense and space electronics, automotive electronics, and high-performance computing. Although the literature reports numerous applications implemented on FPGAs, only very few of them were parameterized. This is mainly due to the difficulty that parameterization adds to the complexity of implementing large applications on FPGAs. In this work, we implement a highly parameterized Particle Image Velocimetry (PIV) application on reconfigurable hardware. The high degree of parallelism in reconfigurable hardware offers an affordable solution to achieve considerable speedup of image processing applications in general and PIV in particular.

PIV is an optical technique for computing the local speed and direction within a dynamic fluid by imaging. It consists of comparing pairs of images of the flow, to which tracer particles are added and illuminated by a laser [3]. The computational algorithm used for this method is known in digital image processing as prototype matching, performed by the cross-correlation function. Because of its characteristics and its non intrusive quality, PIV is widely used in many applications related to fluid flow investigation including turbulent flow studies, jet aircraft performance analysis, bioengineering and medicine. However most of these applications use software to perform the computations, which makes them too slow to achieve real-time processing. Consequently, such implementations cannot satisfy the requirements of many applications where time response is crucial. Not only is the previous work of accelerating PIV very limited, but each implementation targets specific parameters, which restricts them to a limited selection of PIV applications that use the same parameters. Accelerating a system as complex as PIV on reconfigurable hardware is a task that

demands a great deal of effort, which is limited when applied to a PIV application with fixed parameters.

We present the first highly parameterized digital PIV system implemented on reconfigurable hardware. The parameterized circuit has been designed and implmented on an ADM-XRC-5LX FPGA board from Alpha-Data. The computational speed of the hardware presents an average of 50x speedup in comparison with a software implementation running on a 3 GHz PC. The speedup of the PIV on reconfigurable hardware is mainly due to the perallelism and the deep pipelining of the components building the circuit, as well as the custom memory interface designed to optimize memory intefacing.

1.2 Contributions

In this research, we investigate the function and the implementation of a highly parameterized, real-time PIV system using a FPGA board. The contributions of this work are:

- A highly parameterized digital PIV system is designed using VHDL.
- A highly parameterized digital PIV is implemented on a FPGA board.
- A library of parameterized VHDL components has been built.

This system is unique because it is parameterized to support many different PIV setups. To the author's best knowledge, this is the first highly parameterized digital PIV system implemented on reconfigurable hardware reported in the literature.

1.3 Dissertation Structure

The remainder of this dissertation is structured as follows:

Chapter 2 presents the Background of this work. The first section includes an overview of the PIV experiment as well as its algorithm. The second section effers an overiew about reconfigurable hardware.

Chapter 3 documents Related Work. First, it presents a summary of ways of modern computing. Then, it documents important work performed using FPGAs. It presents as well related work to the PIV application.

Chapter 4 covers important issues related to the design of the PARPIV, our parameterized PIV on FPGA system, its architecture and challenges of the implementation of a such design.

Chapter 5 presents aspects of hardware implementation and shows the acceleration results obtained from our implementation over a PIV software version.

Chapter 6 presents conclusions of this research and future work that can be conducted based on this dissertation.

Chapter 2

BACKGROUND

In this chapter, we present an overview of the PIV technique and reconfigurable hardware. These two elements are the basic components of this dissertation. First, an introduction to the PIV experiment along with the mathematical background used in the computations are provided.

2.1 Particle Image Velocimetry

Particle Image Velocimetry (PIV) is an optical technique for computing the local speed and direction within a dynamic fluid by imaging [3-6]. The results are two dimensional speed vectors of an observation area. The PIV technique consists of comparing pairs of images of the flow to which tracer particles have been added and illuminated when

imaging. The computational algorithm used for this analysis is known in digital image processing as prototype matching, performed by the cross-correlation function. PIV was introduced by Adrian [4] who conducted a lot of research into the experimental setup of this technique. PIV is applied to many applications including turbulent flow studies, jet aircraft performance analysis and micro-PIV for surgery. It is also opening up new areas in aerodynamic research.

The PIV system consists of both an experimental set-up and a computational platform. In the following we will introduce both these components.

2.1.1 Experimental Set-up

Experimentally, tracer particles are added to the flow, and then illuminated by laser pulses while image pairs of the area of interest are taken by high speed digital cameras with a CCD (Charge Coupled Device) chip. The images record the particles' motion. The seeded particles are carefully chosen to faithfully follow the dynamics of the fluid being studied because the velocity information is extracted from the velocity of these particles and it is assumed that the particles acquire the same dynamic as the fluid. The laser used in the experiment should be high power to provide enough illumination for the particles. Usually double pulsed ND:YAG or copper vapor lasers are used. An optical system built of mirrors and lens is placed to convert the laser light to a thin light sheet to illuminate a plane within the flow. The pictures taken are delivered to a computational algorithm, which by means of cross-correlation techniques, finds similar parts from different pictures and consequently determines the displacement of the particles. Figure 1 shows an experimental set-up for PIV.

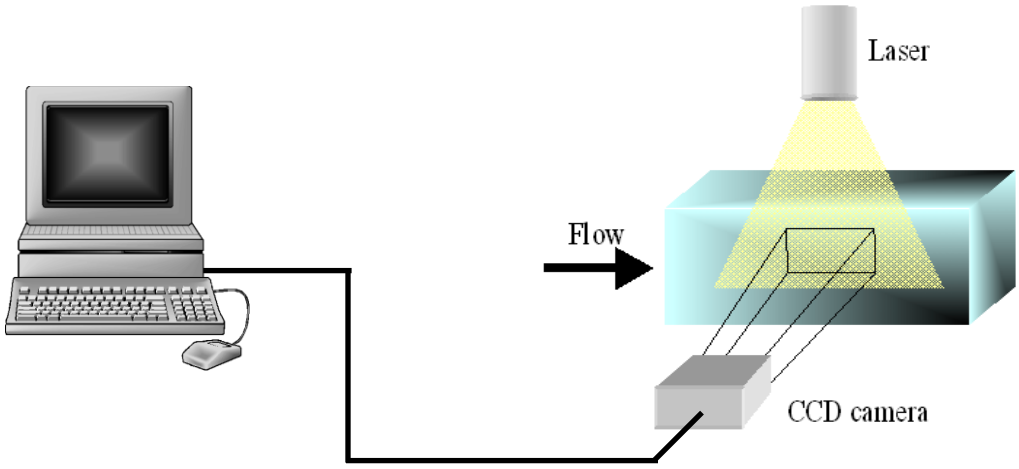

Figure 2.1 PIV experimental set-ups

Although this research does not focus on the experimentation aspect of PIV as it does on the computational algorithm, here we mention some characteristics of PIV experimentation [3]:

• **Nonintrusive**: PIV is an optical method so it does not disturb the fluid as do other techniques which use probes like pressure tubes.

• **Indirect measurement**: PIV measures the velocity of the fluid indirectly because it is considered equal to the velocity of the tracer particles. The density and the volume of the particles are carefully chosen so the particles can faithfully follow the motion of the fluid. The smaller the particles are, the better they follow the fluid movement.

• **Illumination**: The PIV technique requires a source of illumination toward the fluid so the scattered light from the particles offers an exposure to the photographic film or the video sensor. The larger the particles are, the better the light is scattered. However, this is in contrast to the fact that small particles follow the flow of the fluid better. Consequently, a compromise has to be determined depending on the application.

• **Illumination pulse**: The duration of the illumination should be as short as possible in order to freeze the particles during the exposure and to avoid getting a blurred image.

• **Time delay between illumination pulses**: This time should be long enough to be able to determine the displacement of particles but short enough to avoid computing any out-of-plane motion of the particles.

• **Distribution of particles in the flow**: a homogeneous distribution of medium density is preferred for high quality recording to get optimal evaluation.

• **Repeatability of evaluation**: The information obtained from a PIV experiment is fully recorded in the images. Consequently, the results can be post-processed many times and with different algorithms without the need to repeat the experiment.

2.1.2 Mathematical Background for PIV

Statistical Correlation is a concept used in many fields to perform quantitative research and analysis to discern the relationship of co-varying things. It is a mathematical tool that reveals dependence or independence of co-occurring random variables [7-8]. In fact, the correlation was investigated to determine some way of making the units of the variables

comparable, so that the correlation between any two of them can be determined. In other words, the correlation is a measure of co-variation, of how much two things change together or oppositely. The correlation is a strong tool that is used in a variety of scientific and engineering areas. The cross-correlation function is used in signal processing to measure the similarity between two signals or to find the characteristics of an unknown signal by comparing it to a known one. It is also called the sliding dot product and has application in pattern recognition and crypto analysis.

2.1.3 Matching by Correlation

In digital image processing [9], correlation is usually used to find matches of subimage $w(x, y)$ of size J x K within an image $f(x, y)$ of size M x N, where it is assumed that $J \leq M$ and $K \leq N$.

The correlation between f(x, y) and w(x, y) is

$$c(s,t) = \sum_{x} \sum_{y} f(x,y)w\ (x-s, y-t) \tag{2.1}$$

where s = 0, 1 …M-1, t = 0, 1 …N-1, and the summation is taken over the image region where f and w overlap. We assume that the origin of image f is at its top left and the origin of image w is at its center as shown in Figure 2.2. For each pair (s, t) inside image f, Equation (2.1) yields one correlation value. As s and t move inside f, $w(x, y)$ moves around the image area, defining the values of the function $c(s, t)$. The maximum of $c(s, t)$ indicates the position where $w(x, y)$ best matches with $f(x, y)$.

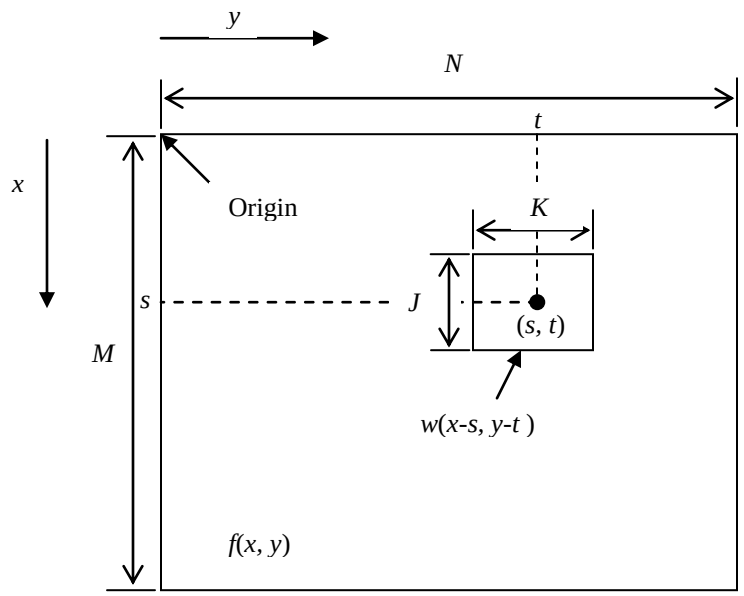

Figure 2.2 Correlation of *f* and *w* at point (*s, t*)

Applying those equations to the gray level of the pixels, they yield:

$$c(x_0, y_0, u_0, v_0) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I_1(x_0 + i, y_0 + j) I_2(u_0 + i, v_0 + j) \qquad (2.2)$$

I_1 and I_2 are the gray level functions of the image (*A*) and (*B*) respectively.

2.1.4 PIV Computational Algorithm

PIV analysis is based on matching by the correlation computation introduced in the previous section. Figure 2.3 provides a description of the PIV method. A pair of images (*A*) and (*B*) of the flow are recorded at times t and $t + \Delta t$ respectively. The images are divided into small areas called "interrogation windows". For each interrogation window from (*A*), a main sub-area $M(x, y)$ with a smaller size than the interrogation window is

positioned in its center. A match for this sub-area is searched for in a corresponding interrogation window from the image (*B*) using the algorithm of matching by correlation. Thus, a current sub-area $C(u, v)$ that is the same size as the main sub-area is cross-correlated with $M(x, y)$. The current sub-area is initially positioned at the top left corner of the interrogation window $I(x, y)$, then slid horizontally and vertically to cover the surface of the interrogation window. The number of pixels by which the sub-area $C(u, v)$ is slid is a parameter in our circuit that is called *Displacement.* The sub-area that yields the maximum correlation to $M(x, y)$ represents a match and thus determines the displacement in pixels of the sub-area taken from the first image. The velocity is the displacement divided by the period of time between the two frames. This process is performed each time the current interrogation window is moved either horizontally or vertically to cover the entire image and consequently the velocity vectors over the entire target space are computed. When a current interrogation window is moved, it overlaps with the previous interrogation window. The number of pixels by which the interrogation window is moved is called the *Overlap*.

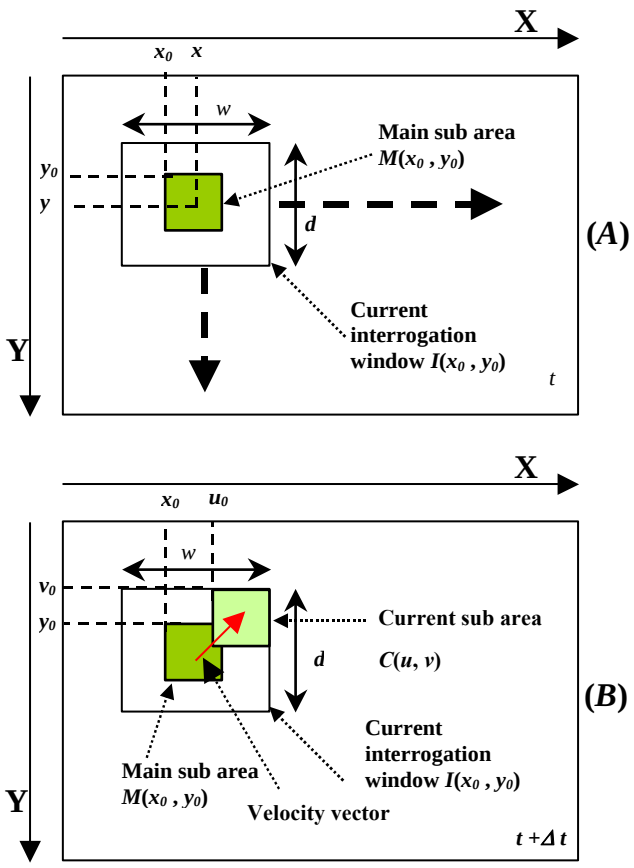

Figure 2.3 Interrogation areas method

2.1.5 Peak Detection

One of the features of digital PIV evaluation is that the position of the correlation peak can be measured to sub-pixel accuracy. Indeed, with 8-bit digital imaging, accuracy can be made to the order of 1/10 to 1/20 of a pixel [3]. The most common methods to estimate the location of the correlation peak are called 3-point estimators that use the correlation values neighboring the maximum correlation to produce a sub-pixel displacement. There are three methods used to perform sub-pixel interpolation: Peak centroid, Parabolic peak fit and Gaussian peak fit. Their formulas are given in [3]. In our project we use the parabolic peak fit expressed by equations (2.5) and (2.6).

$$x_0 = i + \frac{R_{(i-1,j)} - R_{(i+1,j)}}{2R_{(i-1,j)} - 4R_{(i,j)} + 2R_{(i+1,j)}} \tag{2.5}$$

$$y_0 = j + \frac{R_{(i,j-1)} - R_{(i,j+1)}}{2R_{(i,j-1)} - 4R_{(i,j)} + 2R_{(i,j+1)}} \tag{2.6}$$

Here $R_{(i,\ j)}$ is the maximum correlation value and $(i,\ j)$ its coordinates. $R_{(i-1,\ j)}$, $R_{(i+1,\ j)}$, $R_{(i,\ j-1)}$, $R_{(i,\ j+1)}$ are neighboring correlation values.

2.1.6 Pseudo Code for the PIV Algorithm

The following is pseudo code for the PIV algorithm described above.

PIV(A,B)

For each corresponding pair of interrogation windows (I_1) and (I_2) from (A) and (B)

Peak = 0

For each sub-area (S) from (I_2)

Result = cross-correlation of (S) and (M), the main sub area form (I_1)

If result > Peak

Peak = Result

Save the coordinates of the sub area (S_m) yielding the final peak

Perform sub-pixel interpolation for (S_m)

2.1.7 Target Platform

The PIV cross-correlation algorithm is a computationally intensive technique; its complexity increases with the image dimensions as well as the interrogation window dimensions. In addition, the parallel characteristics of this application make it a well suited target for reconfigurable hardware implementation in order to achieve real-time processing.

In the next section, we introduce a review of reconfigurable hardware programming in general then, we discuss the issues addressed in designing a parameterized circuit for a PIV system.

2.2 Field Programmable Gate Arrays

There are several options to implement a computation algorithm, circuit or application. The first option is to write the algorithm as software and run it on a computer with a general purpose microprocessor. The software can be easily modified, but in case of complex computations, software is usually very slow because it executes one instruction at a time. The second option offers the best high performance and it consists of implementing the algorithm in an application specific integrated circuit (ASIC). However, ASICs are not only expensive but not flexible at all. The only way an ASIC can be cheap is in the case of fabricating millions of the same chip which is not always required. In the case of a single chip, there is no manufacturer that would agree to make it simply because it would be too expensive. The third option is represented by the use of Multi-core processors and GPUs. However, Multi-core processors don't provide the fine grained parallelism, which is important for imaging applications such as PIV. A GPU implementation needs a host PC and thus cannot be easily interfaced directly to a camera the way an FPGA implementation can. Another option is to use reconfigurable hardware dominated now days by field programmable gate arrays (FPGAs). This solution offers the highest degree of flexibility. FPGAs use fine-grained structures to perform computations of arbitrary bit widths. In contrary, when a program uses bit widths in excess of what is normally available in a host processor, the processor must perform the computations using a number of extra steps in order to handle the full data width. A fine-grained architecture would be able to implement the full bit width in a single step, without the fetching, decoding, and execution of additional instructions. Furthermore, FPGA programming languages (VHDL and VERILOG) allow easy handling of data bit widths.

Reconfigurable hardware is also relatively more affordable, and can reach high performance. In the following section, an overview of FPGAs is presented as well as the state of the art FPGA computing based design.

Field programmable gate arrays (FPGAs) have their origins in complex programmable logic devices (CPLDs). However, there are basically three main differences between them; the first one is in their logic gate densities. CPLDs have up to ten thousand logic gates while FPGAs have several million. The second difference relates to their architecture. CPLDs have restrictive structures consisting of one or more programmable sum-of-products logic arrays. The last difference is that FPGAs includes embedded computational components (such as adders and multipliers) and embedded memories.

Application Specific Integrated Circuits (ASIC) differ from FPGAs in that, on one hand ASICs are fabricated to perform only one function, and on the other hand they are more expensive. FPGAs can be reconfigured to perform different applications at a lower cost. Although application programming might take a considerable amount of time, it is loaded into the FPGA chip in microseconds.

Reconfigurable hardware has gained popularity in many engineering applications where intensive computations are required. Examples of such domains are aerospace, automotive applications, bioinformatics, cryptography, communication and DSP. In fact, FPGAs have become of great interest to the High Performance Computing (HPC) community once FPGAs started to contain sufficient logic resources [10-11]. FPGAs have been shown to accelerate a variety of applications such as automatic target recognition, string pattern matching, transitive closure of dynamic graphs, Boolean

satisfiability, data compression, and genetic algorithms for the traveling salesman problem [12-13].

FPGAs are more effective for applications that use integer arithmetic [10]. However, They began gaining popularity in floating point applications in 2002 with the Virtex II Pro FPGA from Xilinx, which contains up to one hundred thousand logic cells as well as dedicated DSP slices containing 18-bit by 18-bit, two's complement multipliers. The clock on this chip can reach 400 MHz [14]. At the beginning of this research, the most recent FPGA family from Xilinx was the Virtex-5 which contains 330 000 logic cells and a clock that reaches 550 MHZ [15].

2.2.1 FPGA Computing Boards

Several companies make the use of FPGAs relatively easier by integrating the FPGA device in an electronic board with additional resources. Most of these boards are commercials off the shelf (COTS) computing boards that are widely available. Usually, these boards include storage devices such as SRAM or DRAM, and a bus that connects the board to a host computer. The host communicates with the board to load a bitstream, (the file that configures the FPGA device), send and receive data to and from the memories, or communicates simple instructions to the board. The bus is usually a PCI or PCM bus. The bitstream is the final description of the hardware that will be implemented on the FPGA. The next section discusses generating the bitstream in more details. The board is usually sold with a host API written in a high level language such as C/C++. The host API facilitates the programming of the FPGA board. In the next section, an overview of FPGA computing techniques is provided.

2.2.2 Basics of Reconfigurable Computing

A typical reconfigurable hardware design reads input data from the on-board memories or directly from the host computer using the bus, then the FPGA performs the computations desired, after that the output results are returned to the on-board memories or sent directly to the host computer. The performance of an FPGA board is determined by the speed of the bus, the speed of the on-board memory and the speed of the FPGA device. However, given a specific FPGA board, a design can be improved by implementing more parallelism, building a deep pipeline, and optimizing the memory hierarchy. We will discuss these three design approaches.

2.2.2.1 Parallelism

Parallelism consists of performing many operations at the same time. The main reason an FPGA speeds up an application is because of the parallelism it can implement. The more parallelism implemented, the more speedup obtained. However, parallelizing an application is limited by the resources available in the FPGA device. These resources are the number of logic blocks, on-chip memory and embedded components. Consequently, an estimate of the required resources should be investigated since the more resources the FPGA has, the more expensive it is.

2.2.2.2 Pipelining

Pipelining of a computational task consists of dividing this task into many small tasks. These small tasks are performed by computational elements connected in series so the output of one element is the input of the next one. When the computational task starts processing a datum, the datum is received by the first element of the pipeline, and then passed to the next element making the first element available to start processing another

datum and so on. The elements of the pipeline work by processing different data in parallel. Handling data this way increases the throughput of the pipelined circuit when processing a stream of data.

2.2.2.3 Memory Hierarchy

Memory hierarchy plays an important role in reconfigurable hardware design because the speed of accessing data affects the performance of the circuit. Three types of memory exist, an FPGA board can access the memory on the host computer, and this is the slowest memory access since it uses the bus. The second memory is the on-board memory banks (SRAM or DRAM). The third type is blockRAMs which are the fastest memories on the FPGA device itself. A design should take advantage of the blockRAMs to increase performance. A design should use blockRAMs as cache for data to be processed. A typical data flow for reconfigurable hardware designs is illustrated in Figure 2.11.

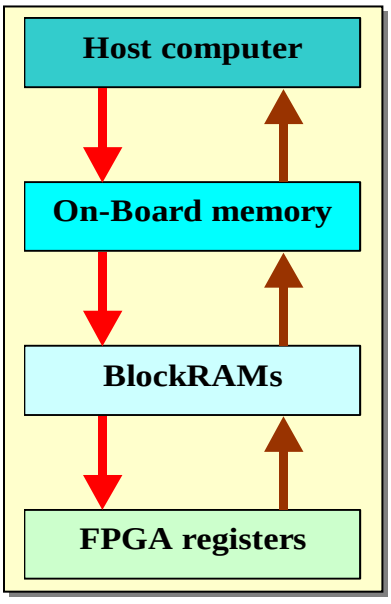

Figure 2.11 Typical FPGAs Data flow

2.2.3 FPGA Design Flow

Reconfigurable hardware design flow usually starts with the description of the coarse grain architecture of the desired circuit. This step is finilized by a fine grain description of the circuit using a hardware description language such as VHDL or Verilog. Then the coded design is simulated and debugged before being synthesised, placed and routed creating a physical layout configured in the FPGA by the bitstream. Figure 2.12 provides an overview of the Xilinx FPGA design flow.

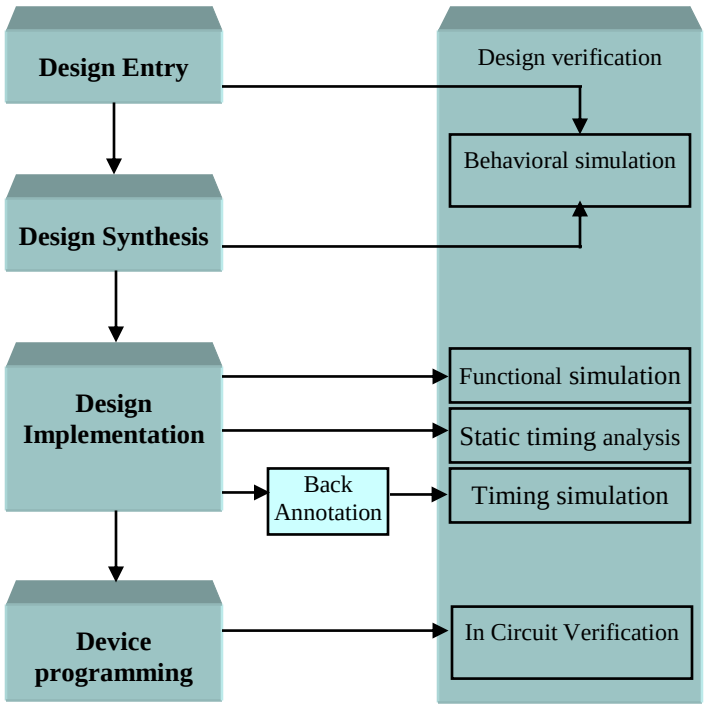

Figure 2.12 FPGA design flow [16]

Design entry consists of developing the circuit design with a hardware description language (HDL), typically VHDL or Verilog. An RTL simulation of the design is performed using CAD tools such as Modelsim to verify the functionality of the circuit with test benches and to correct any bugs.

Design synthesis is a process that checks the syntax of the code and analyzes the hierarchy of the design in order to optimize the architecture. Its goal is to find a trade off between area and delay according to design constraints and the designer's preferences. The result of synthesis is a netlist that include both the logical design and constraints. A netlist can be created as an electronic data interchange format (EDIF) or as an NGC file. Synthesis is done using one of the technology synthesis tools such as Xilinx synthesis tools (XST), LeonardoSpectrum from Mentor Graphics Inc, or Synplify from Synplicity Inc.

Design implementation translates the design into a form ready to be loaded to the FPGA device. It is composed of two main operations, mapping and place-and-route. Mapping consists of fitting the design into the target device by transforming the Boolean logic equations into a circuit of FPGA logic blocks. Placement looks for the specific location of each logic block that is needed for the design. Routing connects the placed logic blocks with each other using the interconnect resources.

Device programming consists of loading the circuit design from a host computer to a target board that contains the FPGA chip. This is done usually in C or C++ using manufacturer API functions.

2.2.4 Target Board

In this research we use the ADMXRC-5LX reconfigurable computing board from Alpha-Data. Figure 2.13 provides a block diagram that illustrates the structure, layout and connectivity of the board. This board has a Virtex 5LX FPGA device from XILINX, and four SDRAM memory banks. A separate Bridge / Control FPGA interfaces to the PCI bus and provides a Local Bus interface to the target FPGA. It also performs all of the board control functions including the configuration of the target FPGA, programmable clock setup and the monitoring of on-board voltage and temperature [17]. DDR2 SDRAM and serial flash memory connect to the target FPGA and are supported by Xilinx or third party IP. The flash memory is used to store the configuration bitstream for the user FPGA. Once the Bridge / Control FPGA is configured, it checks for a valid bitstream in flash and, if present, automatically loads it into the User FPGA.

The board is a PCI mezzanine card. The following are specification of the board [17]:

- Physically conformant to IEEE P1386-2001 Common Mezzanine Card standard
- High performance PCI and DMA controllers
- Local bus speeds of up to 80 MHz
- Four banks of 64Mx32 DDRII SDRAM (1GB total)
- User clock programmable between 20MHz and 500MHz
- Stable low-jitter 200MHz clock for precision IO delays
- User front panel adapter with up to 146 free IO signals
- User rear panel PMC connector with 64 free IO signals
- Programmable 2.5V or 3.3V I/O on front and rear interfaces
- Supports 3.3V PCI or PCI-X at 64 bits

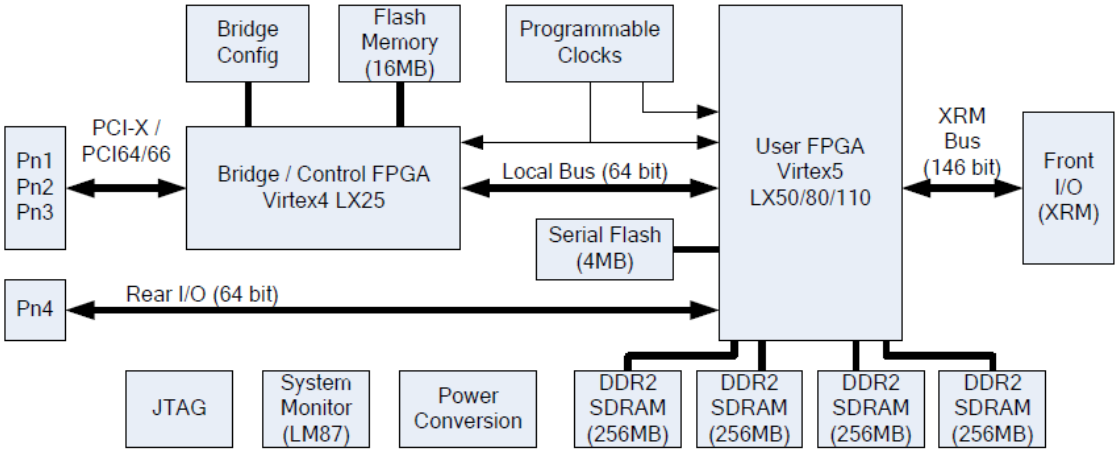

Figure 2.13 ADM-XRC-5LX Block Diagram [17]

In order to use the FPGA board, it is necessary to understand the most important pieces of its architecture, especially those that will be used in our desin:

Local bus

The local bus connects the user FPGA and the bridge; it uses 32 or 64 bit multiplexed address and data path. The bridge design is asynchronous and allows the local bus to run slower or faster than the PCI bus depending on the requirements of the user design. Figure 2.14 provides the interface of the local bus, and table 2.1 lists the interface signals.

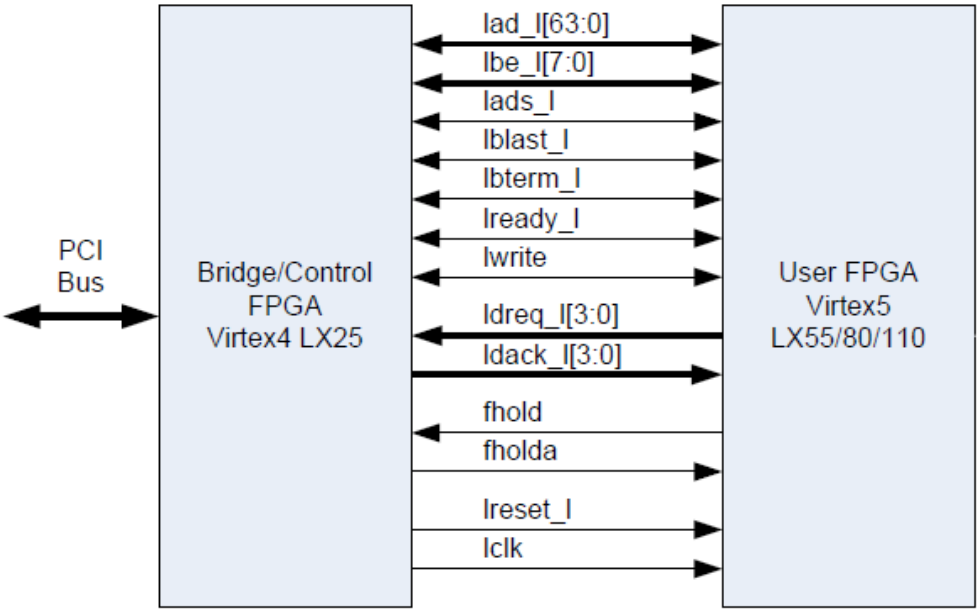

Figure 2.14 Local Bus Interface [17]

Table 2.1 The list of the local bus interface signals [17]

Signal	Type	Purpose
lad[0:63]	bidir	Address and data bus.
lbe_l[0:7]	bidir	Byte qualifiers
lads_l	bidir	Indicates address phase
lblast_l	bidir	Indicates last word
lbterm_l	bidir	Indicates ready and requests new address phase
lready_l	bidir	Indicates that target accepts or presents new data
lwrite	bidir	Indicates a write transfer from master
ldreq_l[0:3]	unidir	DMA request from target to bridge
ldack_l[0:3]	unidir	DMA acknowledge from bridge to target
fhold	unidir	Target bus request
fholda	unidir	Bridge bus acknowledge
lreset_l	unidir	Reset to target
lclk	unidir	Clock to synchronize bridge and target

Clocks

The local bus can run between 32 MHz to 80 MHz and all timing are synchronized to LCLK between the bridge and the user FPGA. LCLK is generated from a 200 MHz reference by a digital clock manager (DCM) within the bridge FPGA. The LCLK is set by writing to the board logic control. The memory clock MCLK can be programmed between 20 MHz and 500 MHz. The PCI interface within the bridge FPGA operates at 66 MHz. In our design, we use both LCLK and MCLK.

Memory banks

The FPGA board has four independent banks of DDRII SDRAM. Each bank consists of two memories in parallel providing a 32 bit datapath. Each bank is 256 MB.

The Xilinx virtex-5

As mentioned previously, our board has a Virtex-5 FPGA from Xilinx. Virtex-5 was introduced to the market in 2006 presenting the most powerful FPGA device [20]. The Virtex-5 architecture is based on the second generation of the Advanced Silicon Modular Block (ASMBL) implemented in the previous generation presented by the Virtex-4 [18]. The Virtex-5 is fabricated with 65-nm copper CMOS process technology and uses 1.0V core voltage.

The main logic resources for implementing sequential and combinational circuits are the configurable logic blocks (CLB). Figure 2.15 provides a digram of a CLB, each CLB contains two slices and is connected to a switch matrix that allows access to the general routing resources.

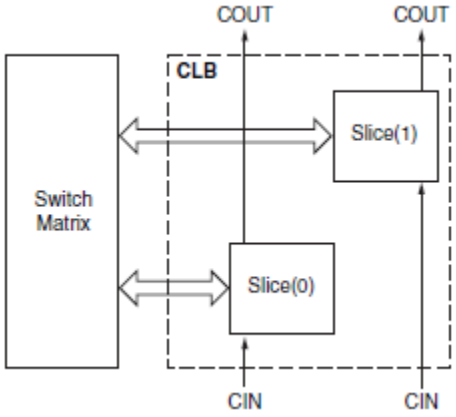

Figure 2.15 Virtex 5 CLB block diagram [18]

The slices are organized differently from previous generations. Figure 2.16 provides the diagram of the Virtex-5 slices. Each Virtex-5 FPGA slice contains four 6-input look-up-table (LUTs), four flip-flops for storage (previously it was two 4-input LUTs and two flip-flops), multiplexers and carry logic. In each LUT there are six independent inputs (A inputs - A1 to A6) and two independent outputs (O5 and O6). The LUT can implement any arbitrarily defined six-input Boolean function. Each LUT can also implement two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs. Only the O6 output of the function generator is used when a six-input function is implemented. Both O5 and O6 are used for each of the five-input function generators implemented. In this case, A6 is driven High by the software. Some multiplexers are used to combine LUT to provide up to eigth-input function generators. Functions with more than eight-inputs are use combined slices to be implemented. Other multiplexers are used to implement fast carry logic.

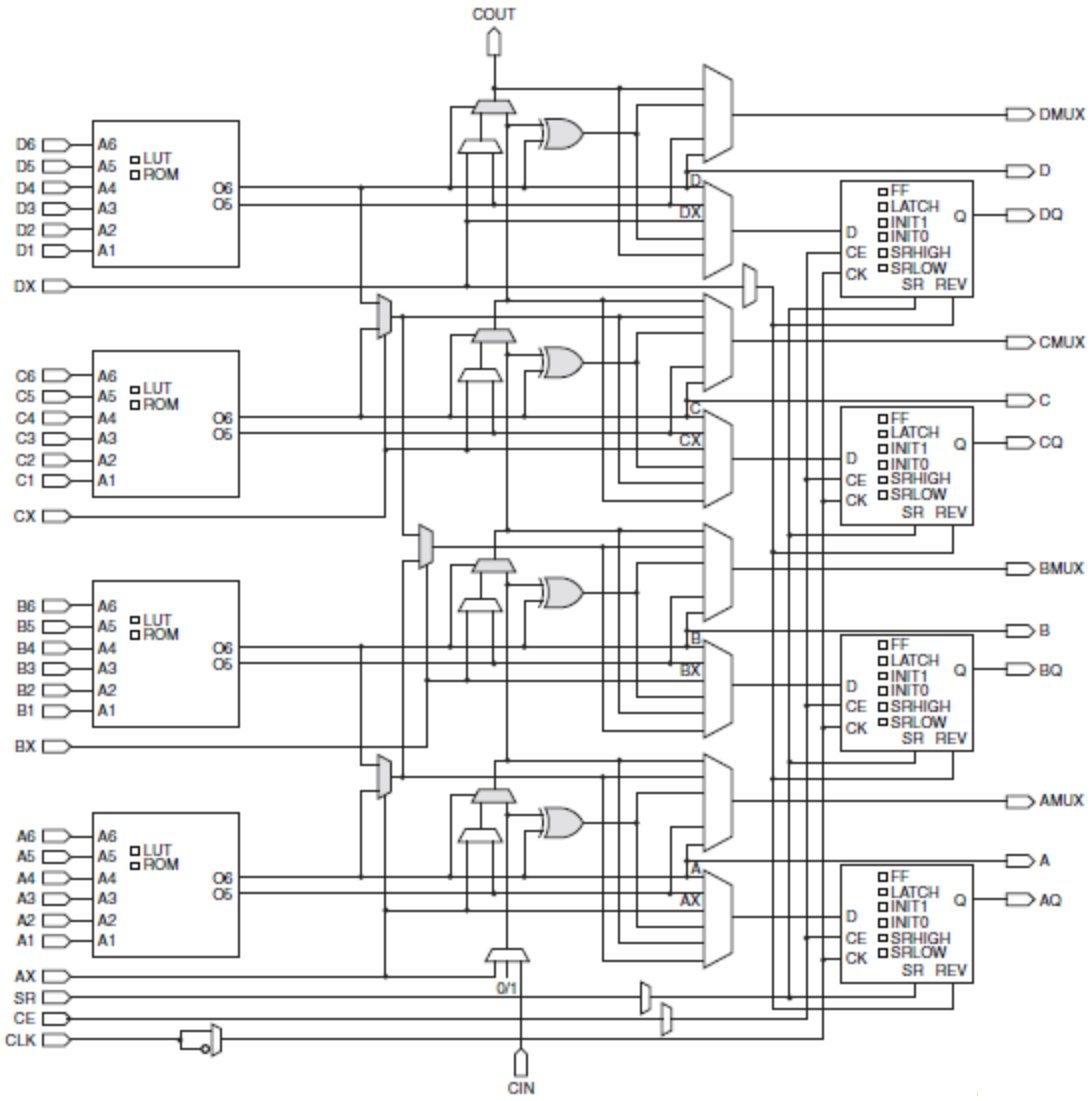

Figure 2.16 Diagram of a slice [18]

The carry chain in the Virtex-5 is used to implement fast arithmetic adders and subtructers. In each slice, the carry chain is four bits wide, and many chains can be cascaded to implement wider arithmetic units. Figure 2.17 shows a diagram of the carry chains carry lookahead logic; there are ten independent inputs (S inputs – S0 to S3, DI inputs – DI1 to DI4, CYINIT and CIN) and eight independent outputs (O outputs – O0 to O3, and CO outputs – CO0 to CO3).

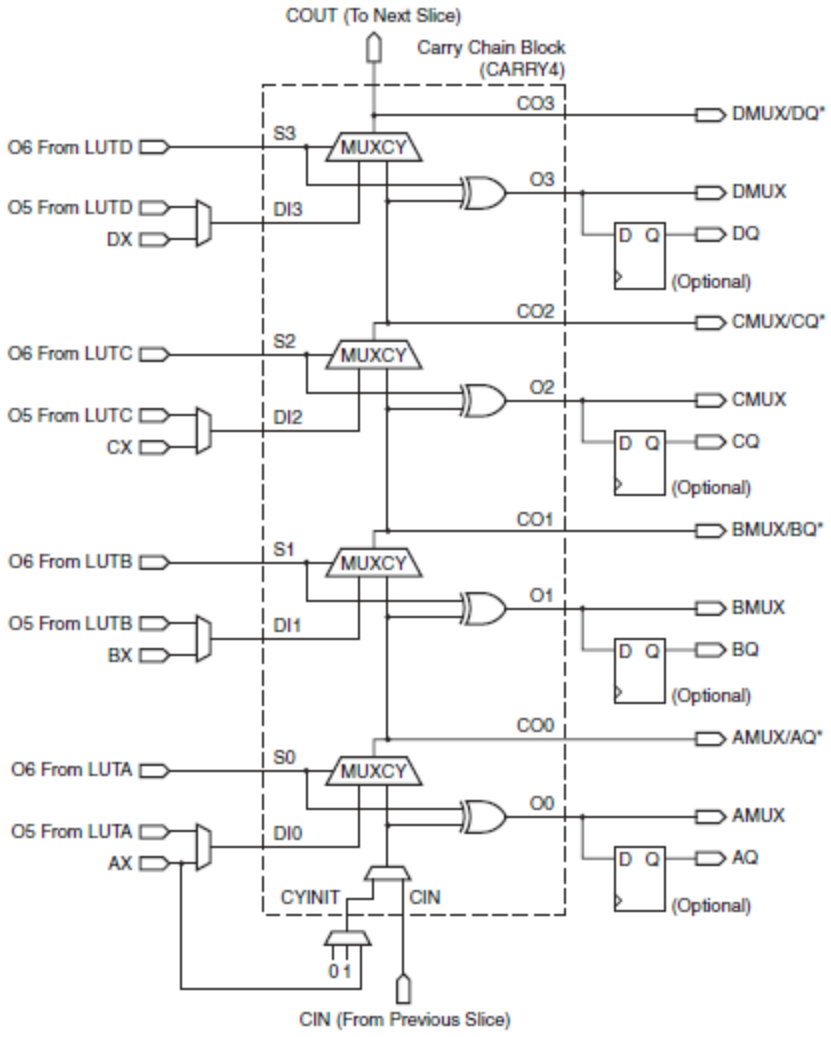

Figure 2.17 Carry chain logic diagram [18]

The S inputs are sourced from the O6 output of a function generator. The DI inputs are sourced from either the O5 output of a function generator or the BYPASS input (AX, BX, CX, or DX) of a slice. CYINIT is the CIN of the first bit in a carry chain. The CYINIT value can be 0 (for add), 1 (for subtract), or AX input (for the dynamic first carry bit). The CIN input is used to cascade slices to form a longer carry chain. The O outputs contain the sum of the addition/subtraction. The CO outputs compute the carry out for each bit. CO3 is connected to COUT output of a slice to form a longer carry chain

by cascading multiple slices. The propagation delay for an adder increases linearly with the number of bits in the operand, as more carry chains are cascaded. The carry chain can be implemented with a flip-flop in the same slice to store the outputs. Each slice contains storage elements that can be used as D-type flip-fops or as latches. More details about storage functions can be found in [18]. Some slices in Virex-5 supports two additonal functions, storing data using distributed RAM and shifting data with 32-bit registers. Distributed RAM are LUTs that are combined to provide storage of large amount of data. They can be configured as single-port or dual-port.

The Virtex-5 contains digital signal processing elements named DSP48E. Figure 2.18 provides a simplified diagram of a DSPE48 slice, each slice contains a 25 x 18 two's complement multiplier, an adder, and an accumulator. More specific informations about the DSPE48 can be found in [19].

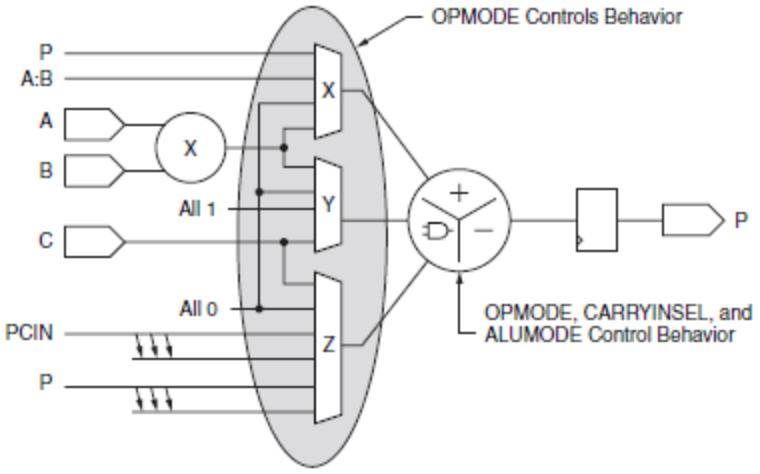

Figure 2.18 DSPE48 slice diagram [19]

Block RAMs are fundamentally 36 Kbits in size. Each block can also be used as two independent 18-Kbit blocks. The Virtex-5 can support a clock frequency up to 550 MHz. Each clock mangaement tile (CMT) contains two digital clock managers (DCMs) that recondition the clock signal by multiplication, division, or phase shifting in order to eliminate the clock skew, the CMT includes also one phase lock loop (PLL).

The Virtex-5 family includes five platforms offering different ratio of features to satisfy the requirements of a broad range of digital design categories. However, we will focus on the features of the device we are using in this work which is the Virtex-5 (XC5VLX110) intended for High-performance general logic applications. A list of the available logic resources in the XC5VLX110 FPGA device is given in Table 2.2.

Table 2.2 Logic Resources in XC5VLX110

Device	Configurable Logic Blocks (CLBs)			DSP48E Slices	Block Ram Blocks			CMTs	Total I/O banks	Max user I/O
	Array	Slices	Max Distributed RAMs		18 Kb	36 Kb	Max Kb			
Xc5vlx110	160x54	17280	1120	64	256	128	4608	6	23	800

2.3 Summary

This chapter presented an overview of the PIV technique which is the application that we will target in this research. We covered essentially the computations involved in this method as well as the algorithm that is used to perform the PIV. We presented a background of reconfigurable hardware based computing. The architecture and principles of FPGA functioning were described along with the design approaches and design flow. Finally, an introduction to the target FPGA board ADM-XRC-5LX was provided

illustrating the most important features of the board as well as those of the Xilinx Virtex-5. In the next chapter, we will present work related to our project.

Chapter 3

RELATED WORK

In this chapter we present an overview of different ways to implement an algorithm in a computing engine. We will cover the advantages and disadvantages of general purpose processors, application specific processors and special purpose processors. We will present important applications that were implemented on FPGAs before covering related work done in the acceleration of PIV.

3.1 Ways of Computing

3.1.1 General Purpose Processors

General purpose processors (GPP), the brain of computers, can perform any computing task. However, although GPP clock rates increased over the last years to achieve magnitudes of GHz, their general architecture has a limitation to accomplish complex computational tasks that might require days, and in some cases, even years to complete on a GPP [21]. This is mainly due to the architecture of general processors which executes instructions sequentially. For each instruction, a GPP needs to perform four steps for each instruction to be executed, *fetch*, *decode*, *execute*, and *writeback* [22]. The *fetch* step consists of retrieving the instruction represented by a sequence of digits from program memory. *Decode*, as the name indicates, interprets the instruction digits according to the GPP defined instruction set architecture (ISA). The *execute* step performs the operation interpreted by the decode. Finally the *writeback* step consists of storing the result into a register or a memory. These steps may be pipelined which is the case in most modern GPPs but the processing is still sequential. There has been tremendous amount of research in computer architecture in general, and GPPs architecture in particular, especially in instruction level parallelism and thread level parallelism. However, the limited parallelism available reduces the performance of GPPs and makes them unable to accelerate important complex applications.

3.1.2 Digital Signal Processors

An alternative to do computations is to use application specific processors (ASPs) dominated by the digital signal processors (DSP) and designed specifically to optimize applications such as those in mobile phones where a large number of mathematical operations are required to be performed on a data set. DSP processors typically have

lower cost and better performance on such applications than GPPs. Popular DSP chips makers include Texas Instrument, Freescale, Motorola, and Analog Devices. The newest DSP processors operates at clock rates on the order of GHz, and perform about several thousands MIPS (million instruction per second) by running several instructions per clock cycle [23, 24, 25, 26]. Top model DSPs support both floating point and fixed point arithmetic. Although DSPs have an improved architecture to increase performance for digital signal processing applications, image processing, and telecommunications, their computations remain sequential which still limits their ability to profit from parallelism in algorithms where speedup can be increased.

3.1.3 Special Purpose Processors

Special purpose processors (SPPs) offer the highest flexibility to the designer to customize the processing of an application. This flexibility allows the implementation of deep pipelining and high degree of parallelism. These possibilities have made it possible to achieve real time processing for many applications. SPPs are represented in three major categories: ASICs, FPGAs and recently GPUs.

ASICs

ASICs are integrated circuits designed and fabricated for a specific use only [27, 28, 29]. Modern ASICs might have over 100 million gates. The process of ASIC design includes many steps: defining the circuit function, RTL descirption language with VHDL or Verilog, logic shynthesis. After that, the circuit is verified by simulation or emulation. Logic synthesis converts the RTL design to a lower level design presented by the

required logic gates and their electrical connections. This set of gates is called gate-level netlist. After that, placement assigns an exact location for the different gates, and routing connects the gates. The output is a file that is used to create a set of photomasks which are used in a semiconductor fabrication lab to produce the physical circuit. Full-custom ASIC design defines all the photo-lithographic layers of the device. While full custom provides higher performance and reduced area, it requires more time for design and manufacturing, high skills, and increased non-recurring cost. Although ASICs offer the highest computing performance, not only are they not flexible but they are expensive because their non-recurring engineering cost might exceed one million dollars for high performance designs.

GPUs

Graphical processor units are SIMD architectures where many little processors work in lock-step. They are processors originally designed to perform graphic applications susch as rasterization, rendering and textures [30-31-32]. However, in recent years, the interest in using these processors to compute general parallel applications has increased significantly. The first GPU was made by Nvidia in 1991. However, porting non-graphical applications to the GPU has increased monumentally since 2003 [33]. Some of these applications achieved great performance speedup and include MRI reconstruction, SQL queries, and stock option pricing. However, in order to optimize the performance on GPUs, the programmer should acquire a deep knowledge of graphics APIs and GPU architecture. Random reads and writes to memory restrict the programming model for GPUs. Until recently, double precision floating point was not supported which elminated

the possibility for a number of scientific applications to be implemented on a GPU. However, NVIDIA came up with CUDA, a software and hardware architecture that allows GPU programmers to several high level programmig languages. GPUs require a long time cycle design to achieve high performance, and having a fixed architecture they offer a limited flexibility to the programmer and thus they are not well suited for all applications especially a parameterized one. FPGAs offer a high degree of flexibility, less than ASICs but much better than GPUs. They are much cheaper than ASICs but more expensive than GPUs. In addition, we can put FPGAs in small embedded packages more easily than GPUs.

3.2 Mapping Applications to FPGAs

Field Programmable Gate Array (FPGAs) have gained a strong momentum in accelerating applications from a broad range of engineering domain. FPGAs provide a generic hardware that is reconfigurable in milliseconds, and thus serve as tools to run many applications. A popular early example is genetic string matching on the Splash machine [34]. The Splash implementation was able to perform the computations approximately 200 times faster than supercomputer implementations. Other applications include mathematical applications such as long multiplication [35], modular multiplication [36], and RSA cryptography [37]. Some physics applications were also implemented on FPGAs such as real-time pattern recognition in high-energy physics [38], Monte Carlo algorithms for statistical physics [39], and Heat and Laplace equation solvers [37]; as well as general algorithms such as the Traveling Salesman Problem [39], Monte Carlo yield modeling [40], genetic optimization algorithms [41], stereo matching

for stereo vision [37], hidden Markov Modeling for speech recognition [42], and genetic database searches [43]. At the Reconfigurable Computing Laboratory at Northeastern University, many applications have been accelerated, A parallel Beam backprojection algorithm which is one of the most fundamental tomographic image reconstruction steps was accelerated to 100 times faster than a software implementation [44]. Most recent work includes prototyping of K-meas clustering for multispectral images where hardware speedup achieves 2150 times faster then software for core computation time [45]. A reconfigurable hardware two-dimensional finite-difference time domain (FDTD) was also implemented showing an acceleration of 24 times over a software implementation [46]. FPGAs represent a strong hardware emulation tool for ASICs design. In fact, often designers of custom chips use FPGAs to verify the correctness of the circuit's behaviour. While software simulation of such complex systems is very slow, logic emulation via FPGAs offers an affordable solution in time and in money for chip verification [13].

FPGAs provide a great potential to accelerate intensive computational applications and achieve high performance. In this research, we take advantage of this potential to implement a highly parameterized digital PIV system.

3.3 PIV Related Work

The term PIV was introduced in the literature by R. J. Adrian in 1984 [2] in a paper where he proposed the distinction between two modes of operation that use optics to compute the velocity of fluids: the laser speckle mode where the illumination of the particles creates a speckle pattern and the mode where the images show individual

particles [2]. Speckle Laser Velocimetry (SLV) was applied to the measurement of fluid velocity fields in 1977 by three research groups: D. B. Barker and M. E. Fourney, T. D. Dudderar and P. G. Simpkins, and R. Grousson and S. Mallick, as described in [2]. Adrian argued that higher particle concentrations were not desirable in fluid dynamics and that one should use particle images rather than speckles. PIV captivated the interest of many researchers who were investigating the structure of turbulent flow. Since then, many experiments and investigations have been conducted to find the appropriate elements for a successful experiment. Many techniques have been applied to perform the interrogation of images. The first technique used for PIV is known as the Young's fringes method, where an optical Fourier transform is used to obtain the power spectrum of the images [1]. Auto-correlation was proposed by Adrian [3]. Later on he provided the theory of cross-correlation analysis [4]. Other techniques were used such as Fourier Transform Method [47], neural networks techniques [48], Lagrangian method [49], and recursive local-correlation [50]. PIV became a whole field that depends on inputs from various engineering fields including the types of illumination, types of coding, types of tracer particles, types of image recording and types of interrogation. The main resolution of these options has led to double-pulsed solid-state lasers, interline transfer PIV video cameras, and interrogation by correlation analysis [2].

The acceleration of the interrogation part of the PIV technique depends on computational technology. In the early days of PIV, the biggest challenge was the interrogation because it is computationally intensive and because the computation technology had limited capabilities. For example, in 1985 fluid laboratories commonly had a digital computer, the DEC PDP 11/23, which typically had 128 Kbytes of RAM

and a 30 MB hard drive. One can guess it was not possible to perform the interrogation by correlation analysis in real time on such computers.

The first implementation of real-time PIV was reported by Engin and Carr [51] where the PIV images were processed by a dedicated vector processing unit that includes a modular structure consisting of an input buffer, correlator unit, and synchronization unit, all of which use programmable electronics. Details of the experimental setup are not available.

A group in Japan [52-53] proposed in 2001 a PIV system based on FPGA processing but with reduced data width because of FPGA limitations at that time. Later, in 2003, the group implemented their design on an advanced FPGA, the Virtex II XC2V6000, with full data width. It is reported that 20 pair of images are compared in one second. The processing uses windows of 50x50 pixels and sub-windows of 30x30 pixels. This implementation meets the requirement of real-time processing by performing the cross-correlation of a window in 447 cycles at 66.5 MHZ clock frequency.

Haiqian Yu [54-55], under the supervision of Dr. Leeser, implemented a real-time PIV system on reconfigurable hardware that uses a smart camera and processes 15 image pairs per second. The processing uses windows of 40x40 pixels and sub-windows of 32x32 pixels. Such dimensions require more processing but result in higher accuracy.

A group in France implemented an adaptive PIV system where only the size of the images was parameterized [56]. Restraining the parameterization to the size of images reveals how difficult it is to parameterize such a complex system. This is confirmed by the few parameterized projects found in the literature. Our system offers much more flexibility by allowing the user to change many parameters of the PIV algorithm

including image size, interrogation window size, sub-area size and overlap between interrogation windows. In addition, for the image size 320x256, our circuit outperform their system [56] by processing 333 pairs of images instead of 204.

Our system is unique because it is parameterized to support many different PIV setups. To the author's best knowledge, this is the first highly parameterized digital PIV system implemented on reconfigurable hardware reported in the literature.

3.4 Parameterization in FPGAs

Parameterized programming is a powerful technique for the reliable reuse and fast generation of reconfigurable hardware circuits. There are libraries that include basic parameterized circuits such as arithmetic units (adders, counters, multipliers, MACs, dividers, square-root circuits, FFT and other components), memory interfaces, multiplexers, decoders. These libraries are usually free or paid sources such as the Core Generator from Xilinx. The literature reports very few research projects that implemented parameterized systems on FPGAs. The rarety of existing parameterized systems is mainly due to the difficulty of designing such systems as well as the constraints that exist in the development tools. There are efforts to upgrade HDLs but they are still in the beginning and it will take time before changes will be made. xHDL is a meta-language defined on top of VHDL [57] it addresses many issues of parameterization but it is still new and not widely used. It is beyond the scope of this work to investigate new language techniques to make parameterization easier but our research can be of great motivation to work on projects in this direction.

3.5 Summary

In this chapter, we discussed different ways to perform computing in general. We presented general purpose processors, application specific processors and specific purpose processors. We described the difference between these categories and why FPGAs present a good target for our application. In addition, we presented previous work done to accelerate the PIV application and what distinguishes our PIV implementation from other implemetations. Next, we will cover in details the design and the implementation of PARPIV, along with acceleration results for circuits implmented with different parameters.

Chapter 4

PARPIV: PARAMETERIZED PIV DESIGN

In this chapter, we present an overview of the high level design of PARPIV, a parameterized digital PIV system targeting an FPGA board. We also discuss specific issues related to the implementation of the parameterized PIV algorithm; we present a review on parameterization in VHDL. Then, we describe in detail the parameters of our PIV system. After that, we present the design architecture of the reconfigurable PIV circuit, some specific challenges are discussed deeply.

4.1 Parameterization

Design reuse is one of the most important reasons of using parameterization. Many different applications use the same components but usually with different requirements. Consequently, it is desirable to be able to customize these components so they can be tuned for different needs. This customization is normally specified by explicit or implicit parameters. The most important parameters in coding hardware circuits is the width of the component, which determines the number of bits of data signals, as in 32-bit adder for exemple. VHDL supports parameterization in many ways. It provides methods to pass parameters into an entity. VHDL uses generic parameters [58-59] whose values are passed in the same way as ports. In addition, packages such as std_1164 and numeric_std allow the user to define unconstrained arrays which are implicitely parameterized. The language also has two language constructs to replicate structures, for generate and for loop. VHDL provides different mechanisms to pass and infer parameters and includes several language constructs to describe the replicated structure [59]. In the following, an overview of these mechanisms and their use is described.

4.1.1 Types of Parameters

In a parameterized design, there are two types of parameters, width parameters and feature parameters. The width parameters are used to specify the number of bits of input and output signals. The size of internal signals are derived accordingly. The widths of data signals can be modified from one design to another to meet different requirements. For example, a parameterized multiplier circuit design can be easily

modified to process 16-, 32- or N-bit operands. The easiness of the modification comes from the fact that the decription of the design logic is independent of the width of the data signals. However, as the design become larger with many parameterized modules, its modification for different bit widths may require a lot of time depending on how much big is the design. A parameterized circuit makes the modification automatically; the cost comes in the process of parameterization.

Feature parameters can be used to specify the structure or organization of a design. We can use them to include or exclude certain functionalities or features from the implementation of a circuit. We can also use them to select a particular version of the design. For example, a feature parameter can specify whether to use a synchronous or asynchronous reset signal for a component. A feature parameter might also be used to choose from several possible implementations of the component. For instance, we can chose between a pipelined version or non pipelined version of a component. Feature parameters are different from width parameters in the way that they require the design of different descriptions of the circuit.

4.1.2 Specifying Parameters

In VHDL, three mechanisms exist to spedify the parameters. We can use generics, unconstrained arrays or array attributes. VHDL uses generic parameters [59] whose values are passed in the same way as ports. Generics behave similar to parameter passing between the main function and a routine in a traditional programming language. The other two methods derive their parameter values indirectly from a signal or from a port declaration. An example of a register whose input width is parameterized is shown in

Figure 4.7 to illustrate the syntax and the use of such VHDL models. The instantiation of this register is done by generic mapping and port mapping as illustrated in Figure 4.8.

```
Entity reg_n is
generic ( n:     integer);
        port (
                D:          IN std_logic_VECTOR(n-1 downto 0);
                Q:          OUT std_logic_VECTOR(n-1 downto 0);
                CLK:        IN std_logic;
                CE:         IN std_logic;
                ACLR:       IN std_logic);
end reg_n;
Architecture Behavioral of reg_n is
Begin
Process (CLK)
        Begin
                if ACLR = '1' then
                Q  <= ( others => '0');
                elsif rising_edge(clk)  then
                        if CE = '1' then
                        Q <= D;
                        end if;
                 end if;
end process;
end Behavioral;
```

Figure 4.7 Parameterized Register Code

```
My_reg : reg_n
            generic map ( n  =>  32)
            port map(
                    D                   =>      A,
                    Q                   =>      B,
                    CLK                 =>      Pclk,
                    CE                  =>      tmp_ce,
                    ACLR                =>      reset
                );
```

Figure 4.8 Instantiation of Parameterized Register

4.1.3 Benefits of Parameterization

Parameterization allows design adaptation and reuse. It is beneficial for both designers and consumers. Instead of designing components from scratch or modifying existing ones to fit the new requirements, reconfigurable hardware designers use parameterized components from libraries that have been simulated and tested; this helps to decrease time to market. For consumers, using parameterized circuits, either small components or large designs, allows the flexibility of changing the circuit parameters and then having new circuits with new parameters in a very short time. Such flexibility is in high demand especially in research areas where intensive computing is required and appropriate computational parameters are still being investigated or simply where application parameters differ from one domain to another which is the case for PIV applications.

4.2 PIV Design Architecture

In this section we will present the details of the our design such as the parameters involved in the PIV Algorithm and the platform architecture, the high level PARPIV architecture and the major challenges faced through the parameterization process.

4.2.1 PIV Parameters

The parameterization of a complex digital system like PIV on reconfigurable hardware is a difficult task and should follow a specific methodology. Our process consists of extracting the parameters first from studying the computational algorithm and the platform used, then specifying the high level architecture that will be implemented to perform the computations. When designing the low level circuit, we need to find the relations between the parameters of the circuit and the different sub-components and

signals of the design. These relations are translated to a number of equations that computes the parameters of the components and the values of the replicated structures inside the design and those inside the components as well. Following this methodology, we examined the cross-correlation algorithm and its implementation on an FPGA board, then we extracted the parameters shown in Table 4.1. These parameters determine the behavior of every component in the PIV system. For example, one of the components we designed is a parameterized and deeply pipelined unsigned multiplier. This is a relatively more complex circuit. The PIV parameters will determine the width of this multiplier. This width will determine the number of replicated adders inside the multiplier, their width and the structure of the reduction tree used inside the multiplier. Appendix A shows the VHDL code of a parameterized and pipelined unsigned multiplier.

Table 4.1: Parameters of PIV Circuit

Img_width	The width in pixels of the images (*A*) and (*B*)
Img_depth	The depth in pixels of the images (*A*) and (*B*)
Area_width	The width in pixels of the interrogation window
Area_depth	The depth in pixels of the interrogation window
Sub_area_width	The width in pixels of the sub-areas
Sub_area_depth	The depth in pixels of the sub-areas
Displacement	Number of pixels by which a sub-area is moved inside an interrogation window
Pixel_bits	Number of bits that represent a pixel
RAM_width	Number of bits in each memory address
Overlap	Number of pixels by which an interrogation window is moved within an image

The PIV parameters can be split into two groups, one that depend on the equipment being used, and the other that is related purely to the computation of the cross-correlation. The dimensions of the images depend primarily on the camera characteristics. Note that these dimensions vary depending on the PIV domain application. These dimensions are the parameters Img_width and Img_depth which respectively represent the horizontal and vertical size in pixels. The RAM_width parameter depends on the width of the memories on the FPGA board. This parameter makes our circuit portable to other FPGA boards. The Pixel_bits indicates the image quantization level (QL) (i.e. bits/pixel). Making this parameter flexible allows our system to integrate different CCD camera technologies that might have different resolutions. This factor is important when single particle images are measured such as in particles tracking velocimetry or super-resolution PIV [1]. The remaining parameters concern the cross-correlation computation. Area_width and Area_depth determine the dimensions of the interrogation windows. Sub_area_width and Sub_area_depth determine the dimensions of the sub-areas whose velocity is computed. The Overlap parameter is a density factor that determines how close the velocity vectors will be to each other.

Specification of the parameters at the top of the PIV reconfigurable circuit configures all the components needed by the architecture to perform its task. The architecture is described next.

4.2.2 PARPIV Architecture

The architecture of PARPIV, our parameterized PIV circuit, consists of a control unit and a data path unit. The control unit includes a main finite state machine (MFSM) that

controls the overall system, a loading FSM that regulates loading of interrogation windows from the memory board to blockRAMs on the FPGA chip, and a correlation FSM that controls communications and synchronizes pipelining of the correlator component. The data path unit is composed of a myriad of components. Different types of counters, adders, multipliers, registers, comparators, block RAMs, and memory interfaces, as well as an accumulator and a divider are used. All of the components are parameterized and pipelined.

The MFSM generates the memory addresses for a pair of corresponding interrogation windows from images (*A*) and (*B*), then commands the LFSM to load them while a previous pair is processed. In this way, no time is wasted waiting for loading data from on-board memories except for the first pair. Parameterized fifos are used to temporarily store addresses to smooth over differences in response from the memories.

Once a pair of interrogation areas is loaded, the MFSM controls modules that generate the on-chip memory addresses for the sub-areas. The MFSM commands the CFSM to start performing all the correlations required to determine the peak. The correlation FSM (CFSM) controls a correlator component that is fully pipelined and includes an optimized reduction tree to speedup the correlation of the sub-areas. The MFSM jointly with the CFSM also make sure that the correlator unit works continually to reduce the global processing time. For each pair of interrogation windows, the correlation results are stored in a blockRAM in order to compute the sub-pixel interpolation that requires the neighboring correlations of the peak. Once this step is completed, the MFSM commands a memory interface module to store the results in on-board memory. Figure 4.10 provides a simplified block diagram of the PIV system architecture.

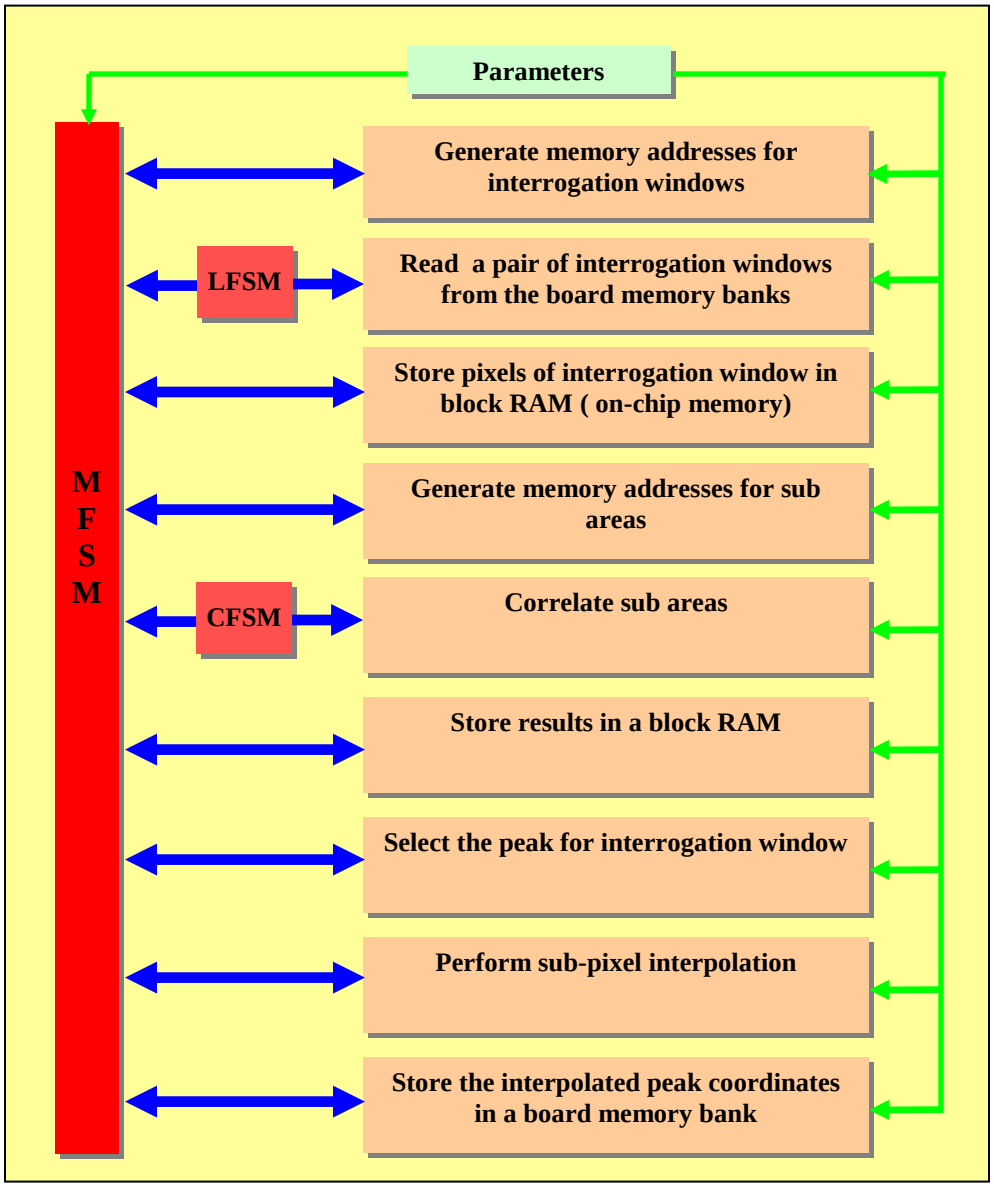

Figure 4.10 Block Diagram of PIV Hardware Circuit

4.2.3 Challenges of Parameterization

While parameterizing relatively small circuits is a straightforward task when the structure of the design requires repetitive structures, it is more challenging for large circuits for two reasons. First, we need to extract all the relations between the input parameters and the parameters of the components that will be used in the circuit. This processing should be done carefully to assure correctness of the design. The second reason consists in the fact that in complex circuits, irregular repetitive nature of the structure might show up because of particular parameters. In this case, the parameterization of these components should solve the design for the most general case, which might prone challenging as will be described later for some examples.

We designed many parameterized components; some of them are basic logic elements such as registers, shift registers, latches, and counters. We designed parameterized components that result in adders, comparators, and blockRAM interfaces. We also designed a parameterized FIFO.

More challenging circuits also had to be designed. For instance, an optimized reduction tree (ORT) is challenging to parameterize. ORT computes the addition of n values in $\log_2(n)$ steps by first adding all the values in pairs, then adding the results is pairs, until the final addition is computed. ORT is straightforward when the initial number of values is a power of two. However, ORT must work in the general case where the initial number of values may be odd or even, power of two or not. Dealing with these different cases makes the ORT more difficult to design. In addition, targeting high performance pushes us to make the ORT completely pipelined, which is even more challenging. Figure 4.11 shows the structure of an ORT in the simple case when n is a

power of two. The long thin rectangles represent pipeline registers and the circles represent two input m-bit adders.

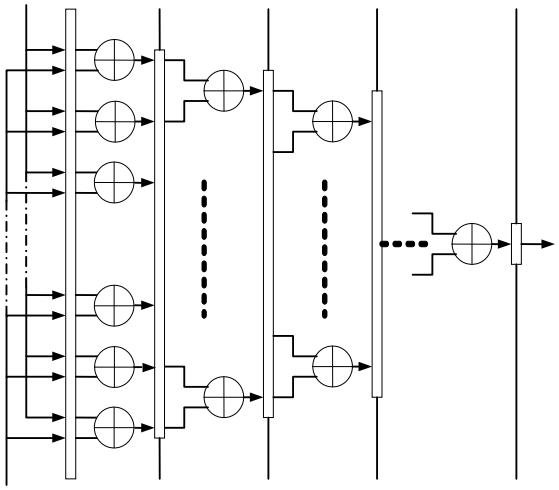

Figure 4.11 Optimized Reduction Tree Structure

Parameterization reveals another issue. We have a finite state machine CFSM that manages the correlation of two interrogation windows. The number of states of the CFSM depends on the number of stages inside the correlator unit. However, parameterization of the circuit yields a different number of stages inside the correlator depending on the values of the parameters. Since VHDL doesn't allow finite state machines to vary dynamically; they should be specifically declared at the beginning of the code. This issue required us to redesign the CFSM and add other logic elements to make the CFSM have a constant number of states independent of the circuit's parameters.

4.3 Sub-Pixel Interpolation

One of the features of digital PIV evaluation is that the position of the correlation peak can be measured to sub-pixel accuracy. With 8-bit digital imaging, accuracy can be

made to the order of 1/10 to 1/20 of a pixel [1]. The most common methods to estimate the location of the correlation peak are called 3-point estimators that use the correlation values neighboring the maximum correlation to produce a sub-pixel displacement. There are three methods used to perform sub-pixel interpolation: Peak centroid, Parabolic peak fit and Gaussian peak fit. Their formulas are given in [1]. In our project we use the parabolic peak fit expressed by equations (1) and (2).

$$x_0 = i + \frac{R_{(i-1,j)} - R_{(i+1,j)}}{2R_{(i-1,j)} - 4R_{(i,j)} + 2R_{(i+1,j)}} \qquad (1)$$

$$y_0 = j + \frac{R_{(i,j-1)} - R_{(i,j+1)}}{2R_{(i,j-1)} - 4R_{(i,j)} + 2R_{(i,j+1)}} \qquad (2)$$

Here $R_{(i,j)}$ is the maximum correlation value and (i, j) its coordinates. $R_{(i-1,j)}$, $R_{(i+1,j)}$, $R_{(i,j-1)}$, $R_{(i,j+1)}$ are neighboring correlation values. To perform sub-pixel interpolation, a parameterized and pipelined divider needs to be designed. We are designing a fixed-point divider based on the Goldschmidt algorithm [60] similar to the one presented in [61-62]. The algorithm to perform the division $q = v/d$ consists of multiplying both the numerator and the denominator by the same sequence of numbers such that the new denominator converges to one. Consequently, the dividend converges to the quotient.

$$q = \frac{v\, x^{(0)}\, x^{(1)} \ldots x^{(n-1)}}{d\, x^{(0)}\, x^{(1)} \ldots x^{(n-1)}} \qquad (3)$$

The method assumes a bit-normalized fractional divisor d in [1/2, 1). If this condition is not fulfilled initially, it can be made to hold by appropriately shifting both v and d. The computation algorithm is defined by the following equations:

$$d^{(i+1)} = d^{(i)} x^{(i)} \qquad \text{with} \qquad d^{(0)} = d$$

$$v^{(i+1)} = v^{(i)} x^{(i)} \qquad \text{with} \qquad v^{(0)} = v$$

$$x^{(i)} = 2 - d^{(i)}.$$

The recurrence equations become:

$$d^{(i+1)} = d^{(i)} (2 - d^{(i)}) \qquad \text{with} \qquad d^{(0)} = d$$

$$v^{(i+1)} = v^{(i)} (2 - d^{(i)}) \qquad \text{with} \qquad v^{(0)} = v$$

The iterations are made n times until $d^{(n)} \approx 1$. The choice of $x^{(i)} = 2 - d^{(i)}$ is done because it offers a quadratic convergence of $d^{(n)}$ to one.

Indeed, $$d^{(i+1)} = d^{(i)} (2 - d^{(i)}) = 1 - (1 - d^{(i)})^2.$$

Thus, $$1 - d^{(i+1)} = (1 - d^{(i)})^2.$$

Since $d \epsilon [1/2, 1]$, $1 - d^{(i)} \leq \varepsilon$, so $d^{(i)}$ is close to 1. $d^{(i+1)}$ will be closer to 1 because $1 - d^{(i+1)} \leq \varepsilon^2$, from whence we observe the quadratic convergence.

Having $$d \epsilon [1/2, 1),$$

Then $$1 - d^{(0)} \leq 2^{-1}$$

$$1 - d^{(1)} = (1 - d^{(0)})^2 \leq 2^{-2}$$

$$1 - d^{(2)} \leq 2^{-4}$$

$$1 - d^{(n)} \leq 2^{-2^n}.$$

If the machine word is *k* bits in length to the right of the binary point, then the closest value to 1 that the machine can achieve is 1- 2^{-k}. The iterations stop when 2^n equals or exceeds *k*. Thus, the number of iterations is $n = \lceil \log_2 k \rceil$.

A complete divider circuit has been designed. Ahigh level block diagram of the divider is provided in Figure 4.12. The norlalization circuit brings the divident and the diviser values to the interval [1/2, 1) and stores the number of shifts and their directions. This information is passed through the pipeline stages to the corrector ciruit, which shifts the quotient obtained from the divider circuit to compute the correct quotient for the original inputs. The whole divider circuit is pipelined to increase the throuput of the design. The circuit is also parameterized and can handle inputs with different widths.

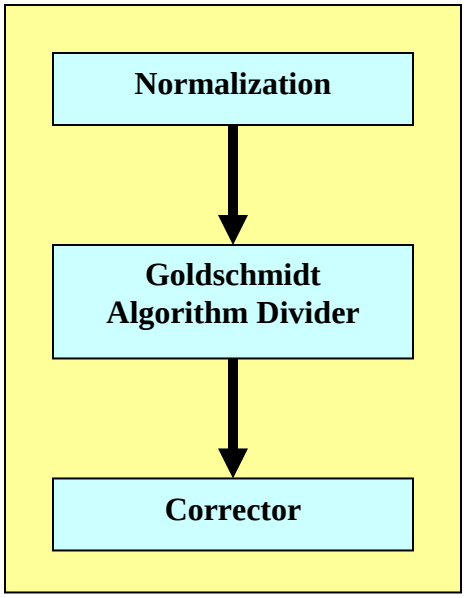

Figure 4.12 Complete divider circuit

As a result of this project, a library of parameterized components has been built. The library includes basic logic elements as well as blockRAMs, accumulators, multipliers, and fixed-point dividers. All the computational cores are pipelined and optimized. This library will help in rapid design of future projects and can be extended to include more components.

4.4 Summary

Through this research, we investigated the parameterization of a complex digital system, and analyzed the different challenges to make a digital hardware design parameterized. The parameterization of big circuits is challenging. Not only does the design of reconfigurable hardware computing circuits requires much more time than the design of equivalent algorithms circuits in software, but the parameterization of such circuits adds another level of complexity, and consequently adds more time to the design. In addition, this task is complicated for digital elements such as pipelined multipliers, pipelined dividers, and finite state machines. However, the extra effort is worth it due to the ease of reuse of these components. In the next chapter, we will cover the implementation and the acceleration results obtained from our design.

Chapter 5

PARPIV IMPLEMENTATION and RESULTS

In this chapter, we present an overview of the implementation and acceleration results of PARPIV. We explore some technical details of the design of the digital PIV on the FPGA board including I/O interface, data transfer, data path, data flow, memory interface and the control state machines. Results that have been obtained so far are also provided.

5.1 Overview of the PARPIV Hardware Design

The hardware design of the parameterized PIV targets the FPGA board adm-xrc-5lx from Alpha-Data. Thus, the design of the circuit must respect the limitations of the resources available on that board and should use the architecture well. However, since most FPGA

boards have similar architectures, the design is portable with minimum modifications to the host computer interface and use of the on-board memories. Our design uses one Xilinx Virtex-5 FPGA and four on-board SDRAM memories; two memories are used for the input data and two for the output results. The digital PIV circuit is implemented in the FPGA device. The input data is transferred from the host computer to the on-board memories via the PCI bus using a Direct Memory Access (DMA) engine. We plan to use a camera link to send data directly to on-board memory but this step is still under investigation.

DMA is an efficient way to transfer a block of data from or to the host computer with a minimum burden on the CPU. To perform a DMA transfer, the CPU first programs the PCI device's registers where to transfer the data, how much data to transfer and which direction the data should travel in. It then starts the DMA transfer, and typically, the CPU is interrupted by the device once the transfer has been completed. Once data is transferred to the on-board memories, a *start* signal is asserted to notify the PARPIV circuit to start. The PIV circuit reads data from two on-board memories to two blockRAMs inside FPGA device, performs the computation on these data. At the same time another read starts in parallel and caches the data in two other blockRAMs. When the computations of the data are complete they are stored to two on-board memories. When all the computations are finished, the results are sent back to the host computer. Figure 5.1 provides a diagram of the high level PARPIV on an FPGA board.

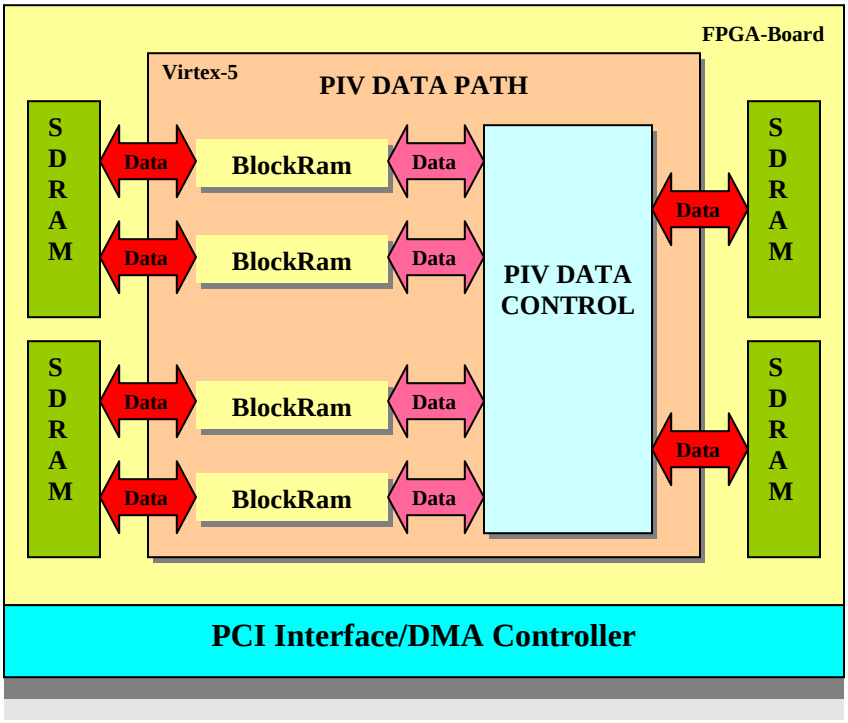

Figure 5.1 FPGA based PIV hardware design

5.2 Memory Hierarchy

The actual design uses three levels of memory as presented in Figure 5.2: (1) The memory in the host computer, (2) the memories on the FPGA board, and (3) the blockRAMs, memories on the FPGA chip. In the following, we will discuss in detail each level, the latency required to retrieve or write data to each type of memory, as well as the mechanisms used to access each one of them. We will discuss the on-board memories and the interface to access them. Finally, we will discuss the blockRams embedded in the FPGA device.

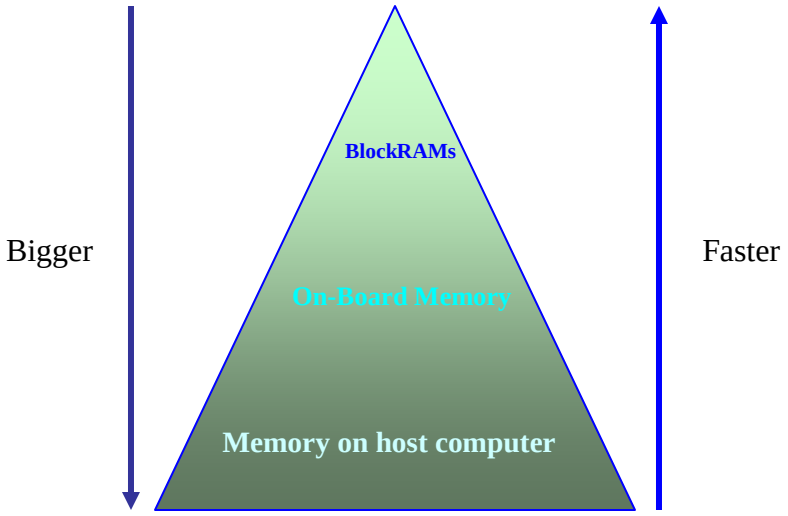

Figure 5.2 Memory Hierarchy for an FPGA Board

On the FPGA board from ADM-XRC-5LX, the DMA transfer is performed by the PCI bus (PCI9656), dedicated hardware from PLX technology. Software runing on the host computer can use these DMA engines for the quick transfer of data by calling API functions such as ADMXRC2-DoDMA and ADMXRC2-DoDAMImmediate.

5.2.1 On-board Memories

The images from the PIV experiment are initially stored on the host computer. These images are transferred to on-board memories on the FPGA card. The FPGA board has four DDRII SDRAMs on-board memories. Each memory is 256 Mbytes making a total of 1 GB of on-board memory. These memory banks are independent and can be accessed either from the host or from the FPGA device. At the highest level of abstraction, the on-board memory can be seen as shown in Figure 5.3. The local bus interface enables the

CPU to read and write to the memory banks. At the same time, the application on the FPGA can also read and write to the memory banks.

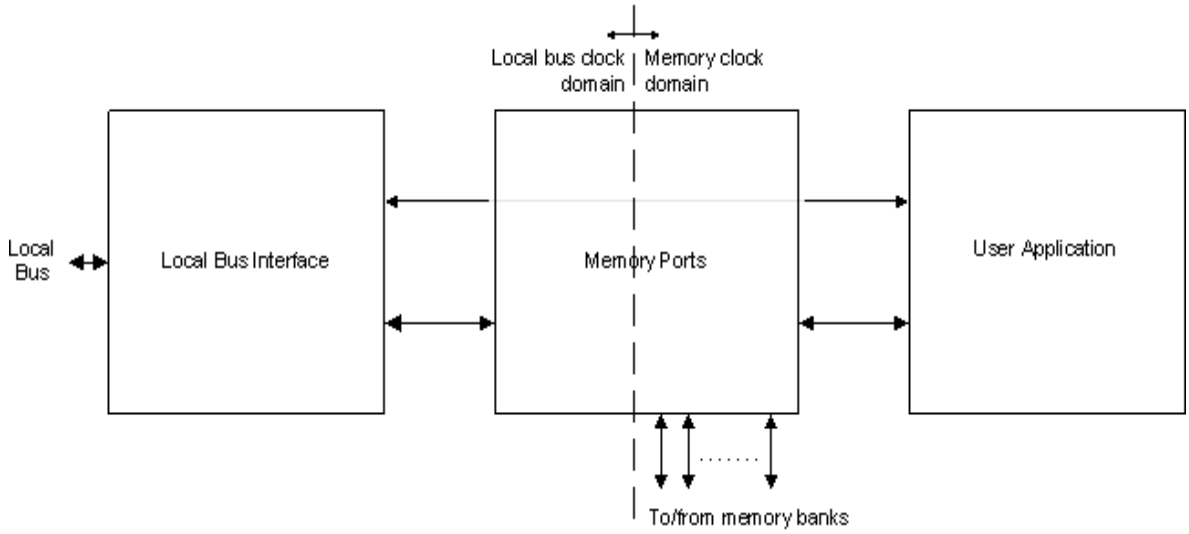

Figure 5.3 Diagram of FPGA board communication [17]

5.2.2 On-chip Memories

Due to the nature of the PIV algorithm, data from interrogation areas are used many times to process the different sub-areas among each interrogation area. Therefore, memory hierarchy is preferred to cache the interrogation areas. For this purpose BlockRAMs are used to store pairs of interrogation area. BlockRAMs are programmable memories which are embedded inside modern FPGA chips. They are more flexible in terms of programmability, and they can be configured with different depths and widths to fit the requirements of the hardware design. This flexibility is of great use for our parameterized design; it allows us to configure BlockRAM units that match the size of interrogation areas. BlockRAMs are also fast in terms of latency, with only one clock cycle delay, and so they are used to build cache modules which read from and write to off-chip memories

continuously and feed data to the computing cores. The Xilinx Virtex-5 (XC5VLX110) has 128 BlockRAM units of 36 Kbits each, and they can operate up to 550 MHz. Note that we would store the whole images to the BlockRAMs if there were enough space but their capacity is limited.

5.3 PARPIV Implementation

We implemented the PARPIV circuit on an FPGA Board introduced in chapter 2, the ADM-XRC-5LX whose block diagram is provided in Figure 5.4. The board, from Alpha-data, includes a Virtex 5 FPGA and four banks of 64Mx32 DDRII SDRAM [17]. The Virtex5 (xcv5lx110), used in our board, uses 6-input look-up table (LUT) technology, it includes 36-Kbit block RAM/FIFOs, and contains 64 Advanced DSP48E slices that includes dedicated multipliers and accumulators [18-20].

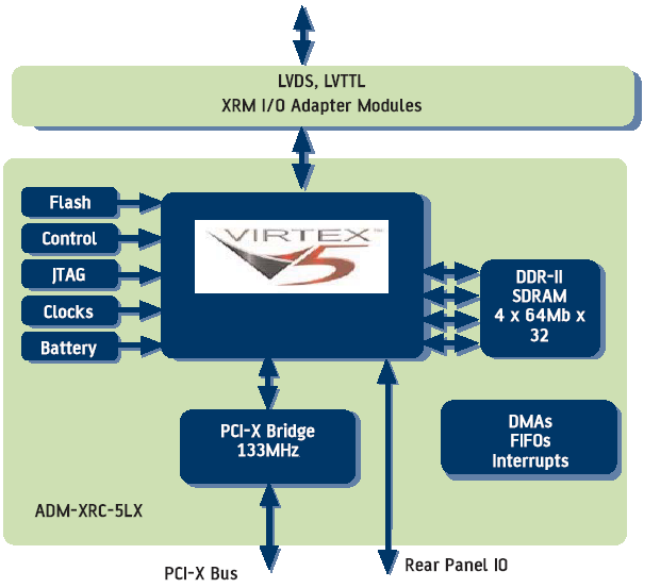

Figure 5.4 ADM-XRC5LX PCI Block Diagram [17]

We used Xilinx ISE 10.103 to perform synthesis, place and route, and generate the bitstream for our design. The board is programmed by a C++ interface assisted by an API library provided by Alpha-Data. Our interface configures the FPGA with the bitstream, loads the images from the host computer to the on-board memories, sends a start command to the circuit and finally returns the results, when they are ready from on-board memory to the host computer.

Table 5.1: Parameters of different circuits

Parameters	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
Img_width	320	512	1024	1200	1600	1024	1024	1024	1024	1024
Img_depth	256	512	1024	1200	1200	1024	1024	1024	1024	1024
Area_width	40	40	40	40	40	24	32	40	48	56
Area_depth	40	40	40	40	40	24	32	40	48	56
Sub_area_width	32	32	32	32	32	16	16	16	16	16
Sub_area_depth	32	32	32	32	32	16	16	16	16	16
Displacement	1	1	1	1	1	1	1	1	1	1
Pixel_bits	8	8	8	8	8	8	8	8	8	8
RAM_width	64	64	64	64	64	64	64	64	64	64
Overlap	20	20	20	20	20	12	16	20	24	28

Table 5.1 provides examples of circuits implemented by our system, and table 5.2 shows statistics about the computations required by every circuit. Note that the parameters of Circuits C1, will result in 81 cross-correlations per interrogation window. Each correlation performs 1024 multiplications. Circuits C2,…,C5 perform the same number of cross-correlations per interrogation area as C1 because they have the same interrogation window size and the same sub-area size. However, the larger the images are, the larger the required processing is. This is noticeable in the results shown in Table 5.3. The circuits, C6,…,C10, have the same image size and the same sub-area size, but the interrogation window size increases from 16 to 56. The number of cross-correlations

increases from 81 to 1681 respectively. Each cross-correlation performs 256 multiplications. Note that every correlation requires data movement.

We first implemented the circuits without integrating sub pixel interpolation, for this experiment, Table 5.3 provides the measured speedup results for the circuits with the parameters defined in Table 5.1. For these circuits, we achieved 17x to 65x speedup in hardware over a standard software implementation in C++ on a 3 GHz Intel XEON microprocessor. This can increase or decrease depending on the parameters of the circuit. Furthermore, The performance results provided in Table 5.2 are obtained using only one correlator component. Performance can be increased by duplicating the correlator unit and slightly modifying the data control unit. The first circuit C1 can process 333 pairs of images per second. Note that this circuit processes images with the same size as the circuit presented in [56], however, our design performs better by processing 333 pairs per second instead of 204. Furthermore, on our board we are using SDRAMs, using SRAMs on the board will boost the performance of the circuit even more.

The running time of the different circuits depends on how much processing is done to compute the velocity vectors. The amount of processing is determined essentially by two factors; the data movement and the number of multiplications that are required from each circuit. Data movement depends on the size of the images, the interrogation area size, and the difference in size between the interrogation window and the sub-area. The larger this difference, the larger the number of movements of the sub-area inside the interrogation area, and the larger the number of multiplications. For example, C5 and C10 have the same size images, but C10, requires a lot more data movement and multiplications.

Table 5.2: Computational statistics of different circuits

Circuits	Interrogation windows	Sub areas in Interrogation window	Cross correlations	# of points
C1	192	81	17492	32x32
C2	625	81	56875	32x32
C3	2601	81	236691	32x32
C4	3600	81	327600	32x32
C5	4800	81	436800	32x32
C6	7225	81	236691	16x16
C7	4096	289	751689	16x16
C8	2601	625	1625625	16x16
C9	1764	1089	2832489	16x16
C10	1296	1681	4372281	16x16

The circuits implemented on the Alpha-Data board use the hardware resources listed in Table 5.4. Note that the amount of resources required is determined by the size of interrogation window and sub-area. Table 5.4 shows that each circuit uses less than half the hardware available in the Virtex-5. This fact opens the door for us to implement more parallelism in our design and thus reach higher performance.

Table 5.3: Speed of different circuits

Circuits	Hardware latency (sec)	Software latency (sec)	Speedup	Pairs per second
C1	0.003	0.17	56	333
C2	0.01	0.65	65	100
C3	0.04	2.2	55	25
C4	0.07	2.94	42	14
C5	0.09	3.90	43	11
C6	0.069	1.15	17	14

C7	0.138	2.95	21	7
C8	0.188	3.937	21	5
C9	0.219	4.546	21	4
C10	0.247	5.04	20	4

Table 5.4: Used hardware resources by different circuits

Resources	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
Number of slice registers	28909 41%	28909 41%	28909 41%	28909 41%	28909 41%	24603 35%	24664 35%	24083 34%	25044 36%	24167 34%
Number of slice LUTs	26761 38%	26761 38%	26761 38%	26761 38%	26761 38%	21922 31%	23031 33%	23049 33%	24607 35%	24297 35%
Number of LUTS used as logic	21552 31%	21552 31%	21552 31%	21552 31%	21552 31%	17250 25%	18350 26%	18366 26%	19922 28%	19612 28%
Number of LUTS used as memory	3953 22%	3953 22%	3953 22%	3953 22%	3953 22%	3953 22%	3953 22%	3953 22%	3953 22%	3953 22%
Number of BlockRAM/FIFO	34 26%	34 26%	34 26%	34 26%	34 26%	34 26%	34 26%	34 26%	34 26%	34 26%

After integrating sub pixel interpolation, we conducted the same experiments. However, this time, circuits C8, C9 and C10 couldn't fit in our FPGA device. Table 5.5 provides the speedups achieved by the circuits C1 to C7. Table 5.6 provides the hardware resources used for their implementation.

Table 5.5: Speedup of different circuits

Circuits	Hardware latency (sec)	Software latency (sec)	Speedup	Pairs per second
C1	0.003348	0.156	46.59	298
C2	0.0117	0.484	41.36	85
C3	0.0507	2.468	48.67	19
C4	0.0707	3.062	43.3	14
C5	0.0946	3.953	41.78	10
C6	0.0745	1.265	16.86	13
C7	0.1409	2.953	20.95	7

Table 5.6: Used hardware resources by different circuits

Resources	C1	C2	C3	C4	C5	C6	C7
Number of slice registers	50919 73%	50955 73%	50862 73%	50952 73%	50952 73%	44779 64%	54441 78%
Number of slice LUTs	47796 69%	47855 69%	47577 69%	47937 69%	47933 69%	41084 59%	46528 67%
Number of LUTS used as logic	42022 60%	42037 60%	41842 60%	42096 60%	42093 60%	35937 51%	41377 59%
Number of LUTS used as memory	4024 22%	4050 22%	3982 22%	4075 22%	4076 22%	3971 22%	3962 22%
Number of BlockRAM/FIFO	33 25%	33 25%	33 25%	33 25%	33 25%	19 14%	23 17%

We notice that the speedups obtained when performing sub pixel interpolation are slightly lower than previously, and the resources are much higher. This is primarily due to the fixed point divider. The division operation is relatively a long operation when compared to addition or multiplication [62], and consequently requires a big amount of hardware. Note that there are other divider implementations for FPGAs that are smaller

but slower than using Goldschmidt's algorithm. These are bit serial implementations. They present a different tradeoff from the divider we are using [63].

In the case the application doesn't need sub pixel interpolation, an important point is that the architecture of this design can be easily modified to duplicate the data path unit and consequently increase the speed. At least, it would be relatively easy to double the performance by inserting another correlator unit so the first circuit can process 666 pairs of images per second. There is future research in how necessary sub-pixel interpolation is for certain applications, and tradeoffs between sub-pixel interpolation and faster speeds.

5.4 Summary

In this chapter, we presented implementation aspects of the PARPIV. Essentially, we discussed the memory hierarchy of our design and the types of memories used in the implementation. We also presented a high level of the implemented hardware design. We presented the different circuits implemented by PARPIV and the speedup results obtained from running our circuits on hardware over a standard software implementation in C++ on a 3 GHz Intel XEON microprocessor.

Chapter 6

Conclusions and Future work

6.1 Conclusions

The objective of this dissertation is to accelerate a highly parameterized version of digital PIV system on a reconfigurable hardware. As a result, PARPIV, a highly parameterized PIV is proposed and implemented on an FPGA board, ADMXRC-5LX from Alpha-Data. The implemented circuits achieved an average computational speedup of 50 times (From 20 to 65 for the circuits implemented). This speedup can be easily increased by using multiple correlators in the circuit. Another option to dramatically increase the speedup consists of using a board with several FPGAs. The more FPGAs used, the more speedup will be obtained.

The circuit proposed, PARPIV, is highly parameterized. This option will allow the deployment of this implementation by users from different domain applications, conducting experiments with different scales and requiring different parameters. This combination of flexibility and speed makes this research unique in high performance computing in general and for the PIV application in particular. In fact, the success of PARPIV will open the door for other researchers to implement other highly parameterized applications on reconfigurable hardware, applications where both speed and parameterization are required to support different end users.

Through this research, a library of parameterized circuits is built. This library contains, registers, counters, adders, high performance multipliers, and high performance fixed-point dividers.

6.2 Future Work

This work presents a great potential for future projects. It can be extended to different research directions.

First, PARPIV can be optimized further to increase the performance achieved by the implementation. We can insert more parallelism in the circuit. Since we are using only one processing element to perform cross-correlation, there is room to speedup the processing by duplicating the correlator unit. This is possible in the Virtex-5 where an average of 30% of its logic resources are not used by the actual circuits that don't integrate sub pixel interpolation. Another way to achieve more speedup is by using a multi FPGA board. This solution might be very attractive to dramtically reduce processing time. However, it is more expensive in both money and development time.

We look forward to work with the Robot Locomotion Group [64] at MIT to integrate our circuit in a fluid control application. This will involve deploying our PIV implementation so that it can be used for real-time feedback control.

PARPIV is a 2D PIV system. It can be used to investigate and implement a 3D PARPIV circuit, a parameterized PIV implementation that will generate velocity vectors for a volume of a fluid in three dimensional space.

The methodology followed in this research project can be applied to design other highly parameterized circuits. We plan to make our parameterized circuit library available for use by researchers in other projects.

Appendix A

The following is VHDL code of a parameterized and deeply pipelined unsigned multiplier.

```
==================================================================
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.my_pkg.all;

-- unsigned parameterized pipelined multiplier
entity unsigned_mult_n is
                    generic ( n : integer);
                    port ( a :     in std_logic_vector( n - 1 downto 0);
                           b :     in std_logic_vector( n - 1 downto 0);
                           Q :     out std_logic_vector( 2 * n - 1 downto 0);
                           clk:    in std_logic;
                           CE :    in std_logic;
                           done:   out std_logic);
end unsigned_mult_n;

architecture Behavioral of unsigned_mult_n  is

component assign_ce is
                   port( a : in std_logic;
                        clk: in std_logic;
                           b : out std_logic);
end component ;
component unsigned_adder_n_n is
                    generic (n  : integer); -- width
                    port ( a   : in std_logic_vector( n-1 downto 0);
                          b   : in std_logic_vector( n-1 downto 0);
                             Q   : out std_logic_vector(n-1 downto 0);
                             clk : in std_logic;
                             ce  : in std_logic);
```

```
end component;
component latch_n is
                generic (n  : integer); -- width
                port ( D   : in std_logic_vector( n-1 downto 0);
                        Q   : out std_logic_vector( n -1 downto 0);
                        clk : in std_logic;
                        ce  : in std_logic);

end component;

constant numb_add             : integer := f_log2(n);
constant numb_add_m1          : integer := numb_add - 1;
constant numb_stg             : integer := numb_add + 1;
constant numb_stg_p1          : integer := numb_stg + 1;
constant stage0_data_length   : integer:= n;

type   pp_matrix is array (0 to n-1 ) of std_logic_vector( n-1 downto 0);
type   pp_extended_matrix is array (0 to n-1 ) of std_logic_vector( 2*n - 1 downto 0);
type   matrix is array( 0 to numb_add,  0 to n-1) of std_logic_vector( 2*n - 1 downto 0);
type   odd_matrix is array( 0 to numb_add ) of std_logic_vector(2*n -1 downto 0);
type   subarray is array(0 to n-1) of std_logic_vector( 2*n  - 1 downto 0);

signal a_reg, b_reg       : std_logic_vector(n-1 downto 0);
signal pp                 : pp_matrix;
signal pp_e               : pp_extended_matrix;
signal zeros              : std_logic_vector(2*n - 1 downto 0) := ( others => '0');
signal m_stage_data       : matrix ;
signal odd_pipe           : odd_matrix;
signal m_stage_data_ce: std_logic_vector(numb_stg + 1   downto 0);

begin
  process(clk )
          begin
          if rising_edge(clk) then
              if CE = '1' then
                                  a_reg <= a;
                                  b_reg <= b;
               end if;
          end if;
  end process;

partpro:   for i in 0 to n - 1 generate
              pp(i) <= a_reg  when b_reg(i) = '1'  else  ( others => '0');
end generate;
```

```
pp_e(0) <= zeros(2*n - 1 downto n -1+1)&pp(0);
ssdd: for i in 1 to n-2 generate
        pp_e(i) <= zeros(2 * n - 1 downto n -1 + i +1)&pp(i)&zeros(i -1  downto 0);
end generate;
 pp_e(n-1) <= pp(n-1)&zeros(n-1  downto 0);

asert: for i in 0 to n-1 generate
                m_stage_data(0,i) <= pp_e(i);
end generate;

-----------------------------
odd_pipe(0)  <= (others => '0' );
m_stage_data_ce(0) <= ce;
ces : for i in 1 to numb_stg_p1 generate
            ce:    assign_ce port map ( m_stage_data_ce(i-1), clk, m_stage_data_ce(i) );
        end generate;
m_reduction_tree   : for i in 0 to numb_add_m1  generate
        m_even: if (((n)/(pow2(i))) mod 2  = 0) generate
                m_reduction_branch : for j in 0 to  ( ((n)/(pow2(i+1))) - 1 ) generate
                        m_add : unsigned_adder_n_n
                                generic map (n => 2*n )
                                port map (
                                        a    => m_stage_data(i,2*j),
                                        b    => m_stage_data(i,2*j + 1),
                                        q    => m_stage_data(i + 1,j),
                                        clk  => clk,
                                        ce   => m_stage_data_ce(i+1)
                                        );
                end generate m_reduction_branch;
                pipe : latch_n
                generic map ( n => 2*n)
                                port map (
                                        D   => odd_pipe(i),
                                        Q   => odd_pipe(i+1),
                                        clk => clk,
                                        ce  => m_stage_data_ce(i+1)
                                        );
        end generate m_even;

        m_odd: if (((n)/(pow2(i))) mod 2  = 1 )  generate
        m_reduction_branch_odd : for j in 0 to  ((n)/(pow2(i+1)) - 1 )    generate
                        m_add_odd : unsigned_adder_n_n
                        generic map (n => 2*n )
                         port map (
                                a    => m_stage_data(i,2*j),
```

```
                                b    => m_stage_data(i,2*j + 1),
                                q    => m_stage_data(i + 1,j),
                                clk  => clk,
                                ce   => m_stage_data_ce(i+1)
                                );
        end generate m_reduction_branch_odd;
        m_add_pipe : unsigned_adder_n_n
                        generic map (n => 2*n  )
                        port map (
                        a    => odd_pipe(i) ,
                        b    => m_stage_data(i,2*(n/pow2(i+1))) ,
                        q    => odd_pipe(i+1) ,
                        clk  => clk,
                        ce   => m_stage_data_ce(i+1)
                        );

        end generate m_odd;
end generate m_reduction_tree;

 m_add_f : unsigned_adder_n_n
 generic map (n => 2*n )
 port map (
                    a    => m_stage_data(numb_add , 0) ,
                     b    => odd_pipe(numb_add ) ,
                     q    => Q,
                     clk  => clk,
                     ce   => m_stage_data_ce(numb_add + 1)
                    );

done <= m_stage_data_ce(numb_stg_p1);

end Behavioral;
==========================================================
```

References

[1] C. Cadden, "FPGA Market Will Reach $2.75 Billion by Decade's End," available: http://www.instat.com/press.asp?Sku=IN0603187SI&ID=1674

[2] D. McGrath, "FPGA market to pass $2.7 billion by '10," available: http://www.eetimes.com/news/design/business/showArticle.jhtml?articleID=18810 2617

[3] M. Raffel, C. Willert and J. Kompenhans. *Particle Image Velocimetry*. Springer-Verlag, Berlin, Germany, 1998.

[4] R. J. Adrian, "Twenty years of Particle Image Velocimetry," *12th International Symposium on Applications of Laser Techniques to Fluid Mechanics*, Lisbon, July 12-15, 2004.

[5] R. J. Adrian, "Statistical properties of Particle Image Velocimetry measurements in turbulent flow," *Laser Anemometry in Fluid Mechanics III*, pp 115-129, 1988.

[6] R. J. Adrian, "Particle-imaging techniques for experimental fluid mechanics," *Annual Review of Fluid Mechanics*, vol 23, pp 261-304, 1991.

[7] R. J. Rummel, "Understanding correlation," *http://www.mega.nu:8080/amp/rummel /uc.htm#top*, Last accessed Dec 20, 2007.

[8] A. Leon-Garcia. *Probability and random processes for electrical engineering.* Addison-Wesley, New York, United States of America, 1994.

[9] R. C. Gonzales, R. E. Woods. *Digital image processing.* Addison-Wesley, New York, United States of America, 1992.

[10] http://en.wikipedia.org/wiki/Fpga. Last accessed Dec 20, 2007.

[11] Katherine Compton and Scott Hauck, "Reconfigurable computing: A survey of systems and software," ACM Computing Surveys, pp. 171–210, June 2002.

[12] Scott Hauck, "The roles of FPGAs in reprogrammable systems," Proceedings of the IEEE, pp. 615–639, April 1998.

[13] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin and C. Kitchen , "An overview of FPGAs and FPGA programming Initial experiences at Daresbury," Council for the Central Laboratory of the Research Councils, 2006.

[14] "Xilinx,"*http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas /overview/index.htm.* Last accessed Dec 20, 2007.

[15] "Xilinx,"*http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/capabilities/index.htm*. Last accessed Dec 20, 2007.

[16] "Xilinx,"*http://toolbox.xilinx.com/docsan/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm.* Last accessed Dec 20, 2007. Last accessed Dec 20, 2007.

[17] Alpha-Data: http://www.alpha-data.com/adm-xrc-5lx.html. Last accessed Jan 25, 2009.

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

[19] http://www.xilinx.com/support/documentation/user_guides/ug193.pdf

[20] http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/capabilities/index.htm. Last accessed Dec 20, 2007.

[21] Thomas H. Cormen. *Introduction to algorithms*. MIT Press, 2001

[22] R. Bryant, D. R. O' Hallaron. *Computer systems*: *a programmer's perspective*. Prentice Hall, 2003.

[23] "Texas Instruments," *http://www.ti.com,* Last accessed Dec 20, 2009.

[24] "Freescale," *http://www.freescale.com,* Last accessed Dec 20, 2009.

[25] "Motorola," *http://www.motorola.com,* Last accessed Dec 20, 2009.

[26] "Analog Devices," *http://www.analog.com,* Last accessed Dec 20, 2009.

[27] V. S. Bagad. *VLSI design*. Technical Publication Pune. India 2009

[28] R. S. Soin, F. Maloberti, and J. Franca. *Analogue-Digital ASICs*. Peter Peregrinus Ltd, London, United Kingdom 1991.

[29] http://en.wikipedia.org/wiki/Application-specific_integrated_circuit

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum, Eurographics (The European Association for Computer

Graphics) and Blackwell Publishing. Volume 26 *Number 1 pp. 80–113*, March 2007.

[31] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth1, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," ACM Transactions on Graphics, Vol. 27, No. 3, Article 18, August 2008.

[32] Y. Liu, E. Z. Zhang, and X. Shen, "A Cross-Input Adaptive Framework for GPU Program Optimizations," IPDPS, IEEE International Symposium on Parallel and Distributed Processing, May, 2009.

[33] http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[34] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen, "Splash: A Reconfigurable Linear Logic Array," International Conference on Parallel Processing, pp. 526-532, 1990.

[35] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to Programmable Active Memories," in J. McCanny, J. McWhirter, and E. Swartzlander Jr., Eds., *Systolic Array Processors*, Prentice Hall, pp. 300-309, 1989.

[36] S. A. Cuccaro, C. F. Reese, "The CM-2X: A Hybrid CM-2 / Xilinx Prototype," *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 121-130, 1993.

[37] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systems*, Vol. 4, No. 1, pp. 56-69, March, 1996.

[38] H. Högl, A. Kugel, J. Ludvig, R. Männer, K. H. Noffz, and R. Zoz, "Enable++: A Second Generation FPGA Processor," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[39] P. Graham, and B. Nelson, "A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash2," in W. Moore, W. Luk, Eds., *Lecture Notes in Computer Science 975 - Field-Programmable Logic and Applications*, London: Springer, pp. 352-361, 1995.

[40] N. Howard, A. Tyrrell, and N. Allinson, "FPGA Acceleration of Electronic Design Automation Tasks," in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 337-344, 1994.

[41] S. D. Scott, A. Samal, and S. Seth, "HGA: A Hardware-Based Genetic Algorithm," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53-59, 1995.

[42] H. Schmit, and D. Thomas, "Implementing Hidden Markov Modelling and Fuzzy Controllers in FPGAs," *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[43] D. P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays," *Advanced Research in VLSI 1991: Santa Cruz,* pp. 139-152, 1991.

[44] M. Leeser, S. Coric, E. Miller, H. Yu, and M. Trepanier, "Parallel-Beam Backprojection: An FPGA Implementation Optimized for Medical Imaging," Journal of VLSI Signal Processing, Vol. 39 No. 3, 2005, pp. 295-311.

[45] X. Wang and M. Leeser, "K-means Clustering for Multispectral Images Using Floating-Point Divide," IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 151-162. April 2007.

[46] Wang Chen, Panos Kosmas, Miriam Leeser and Carey Rappaport, "An FPGA Implementation of the Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm." Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA2004). pp. 213-222. February 2004.

[47] J. Nakamura, Y. Tomira, and S. Honda, "Real Time Particle Image Velocimetry Using Liquid Crystal Display," *IEEE Technology Conference in Advanced Technologies in Instrumentation and Measurement,* vol 3, pp.1205, May 1994.

[48] I. Grant, and X Pan, "The use of Neural techniques in PIV and PTV Measurement Science and Technology," *Meas. Sci. Technol,* vol 8, pp. 1399-1405, December 1997.

[49] M. Sholl, and O. Savas, "A fast Lagrangian PIV method for study of general high-gradient flows," *35th Aerospace Sciences Meeting and Exhibit,* Paper No. 97-0493, 1997.

[50] D. P. Hart, "Super-resolution PIV by recursive local-correlation," *Journal of Visualization, the visualization Society of Japan,* vol 10, 1999.

[51] E. B Arik, and J Carr, "Digital Particle Image Velocimetry system for real-time wind tunnel measurements". *ICIASF 97, International Congress on Instrumentation in Aerospace Simulation facilities,* Asilomar Conference Center, pp 267-277, 1997.

[52] T. Maruyama, Y. Yamaguchi and A. Kawase, "An Approach to Real-time Visualization of PIV Method with FPGA," *11th International Conference on Field-Programmable Logic and Applications,* vol 2147, pp. 601 – 606, 2001.

[53] Toshihito Fujiwara, Kenji Fujimoto, and Tsutomu Maruyama, "A real-time visualization system for PIV," *13th International Conference on Field-Programmable Logic and Applications,* vol 2778, pp. 437-447, September 2003.

[54] M. Leeser, S. Miller, and Y. Haiqian, "Smart Camera based on reconfigurable hardware enables diverse real-time applications,"*12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines,* FCCM 2004, pp. 147-155, April 2004.

[55] Haiqian Yu, Miriam Leeser, Gilead Tadmor and Stefan Siegel, "Real-time Particle Image Velocimetry for Feedback Loops Using FPGA Implementation," *43rd AIAA Aerospace Sciences Meeting and Exhibit,* 2005.

[56] N. Bochard, A. Aubert and V. Fresse, "An adaptive and predictive architecture for parameterized PIV algorithms," *IEEE International conference on Field Programmable Technology,* pp. 241-244, 2006.

[57] A. Miguel, a. Herroro, and M. Lopez-Vallejo, "xHDL: Extending VHDL to improve Core Parameterization and Reuse," *Advances in Design and Specification Languages for SoCs*, pp. 217-235, 2005.

[58] Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems.* McGraw-Hill, Inc. Hightstown, NJ. 1993.

[59] Peter J. Ashenden. *The desinger's Guide to VHDL*. Morgan Kufmann. San Francisco, CA. 2002.

[60] S. Oberman and M. Flynn, "Division algorithms and implementations," *IEEE Trans. on Computers,* vol. 46, no. 8, pp. 833-854, Aug, 1997.

[61] A. Bennis and M. Tull, "Redundant Binary Based Fixed-Point Divider," *IEEE CCECE/CCGEI* Saskatoon Canada, May 2005.

[62] A. Bennis, "Division and Square-Root Based on Redundant Binary Numbers," M.S. thesis, University of Oklahoma, Norman, OK, April, 2005.

[63] X. Wang and B. E. Nelson. Tradeoffs of designing floating-point division and square root on Virtex FPGAs. In *11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 195–203, April 2003.

[64] "MIT," *http://groups.csail.mit.edu/locomotion/russt.html*, Last accessed Feb 20, 2009.

[65] *http://www.piv.jp/challenge/pub*. Last accessed Dec 20, 2007. Last accessed Dec 20, 2007.

[66] M. Morris and R. Kime. *Logic and Computer Design Fundamentals*. Pearson Education, Inc. NJ 1997.