"2D Convolver Implementation On FPGA and 16-pt FFT Implementation"

A Major Project Report

Submitted in Partial Fulfillment of the Requirements for the Degree of

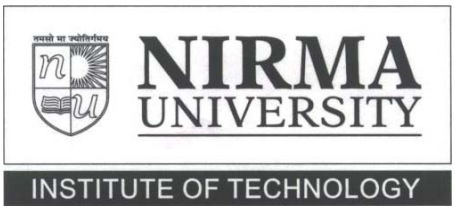
MASTER OF TECHNOLOGY

IN

ELECTRONICS & COMMUNICATION ENGG.

(VLSI Design)

By
NAGARAJU MANCHINENI
(03MEC08)



Department of Electronics & Comm Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382 481

MAY 2005

Certificate

This is to certify that the Major Project Report entitled **"2D Convolver Implementation Of FPGA and 16-pt FFT Implementation"** submitted by **Mr. NAGARAJU MANCHINENI (03MEC08)**, towards the partial fulfillment of the requirements for the award of Degree of Master of Technology in Electronics and Communication Engineering (VLSI Design) of Nirma University of Science and Technology is the record of work carried out by him under my/our supervision and guidance. The work submitted has in my/our opinion reached a level required for being accepted for examination. The results embodied in this major project work to the best of our knowledge have not been submitted to any other University or Institution for award of any degree or diploma.

Date:

Project Guide: **Shri. Rakesh Mehta**
Managing Director
BITMAPPERS
PUNE

Facilitator at Institute: **Prof.Y.N.Trivedi**
Department of ECE
Institute Of Technology
Nirma University of Science and Technology

HOD
Department Of ECE
Institute Of Technology
Nirma University Of Science and Technology

Director
Institute Of Technology
Nirma University Of Science and Technology

Signature of Examiners:

Acknowledge

It gives immense pleasure to express my deepest and most sincere feelings of gratitude to **Mr. Rakesh Mehta** and **Assi. Prof. Y.N. Trivedi** as my thesis supervisors have extensively helped in preparing this project work with their valuable suggestions. The various values that I learnt from them shall remain a source of inspiration for me forever.

My sense of gratitude is also to Dr. N.M devashree, Prof. N. P. Gajjar, Asst.Prof.Pujara, Prof. T. P. and Dr. Desai for their valuable pieces of advice and for the deepinsights given through the various courses they taught.I would like express my sincere thanks to Mr. Veerendra Dhakad, employee of Bit Mapper, who helped me a lot throughout my project period.

Also I am thankful to Mr.Balakrishan Ahirwal, another employee for his comments which made more stronger for pursuing this work. I am grateful to each and every employee of Bit Mapper Integration Technologies Pvt. Ltd Pune, who helped me directly or indirectly for successful completion of my thesis.

I am grateful to Mrs. Poornima Mehta who has given me financial support in the completion of this project. I received lots help from my colleagues Vibhav Shah, Bharti, Shruthi, Magendran. And also I am thankful to my classmates Mahesh, Abhishek, penchal Reddy, Suresh, Sourav, Vishal, Daval, Bavin and Sudhanshu for their cooperation. I thank everyone who made my stay at pune enjoyable. I thank my comp 202.141.82.83 for not crashing during my project period. Special thanks to my parents and my elder brother and my sisters for their tremendous support and love all through. Pune, April, 2005

Contents

List Of Figures

Nomanclatutre

FPGA : Field Programmable Gate Array.

CPLD : Complex Programmable Logic Devices.

CLB : Configurable Logic Block.

IOB : Input Output Block.

FFT : Fast Fourier Transform.

ASIC :Application Specific Integrated Circuits.

DSP : Digital Signal Processors.

XC : Xilinx Chip.

DCT : Discrete Cosine Transform.

DLL :Delay Locked Loop.

PCI :peripheral Component Interconnect.

LUT :Look Up Table.

RAM :Random Access Memory.

PROM : Programmable Read Only Memory.

CE : Clock Enable.

GRM :General Routing Matrix.

BUFT :Tristate Buffer.

JTAG :Joint Test Action Group.

CCLK :Configuration Clock

TDI :Test Data Input.

TAP :Test Access Port.

TDO :Test Data Output.

TCLK :Test Clock

FIR :Finite Impulse Response.

Abstract

Computer manipulation of images is generally defined as Digital image processing (DIP).DIP is used in variety of applications, including video surveillance, target recognition, and image enhancement. Some of the many algorithms used in image processing include Convolution (on which many others are based), edge detection and contrast enhancement. These are usually implemented in software but may use special purpose hardware for speed. With advances in the VLSI technology hardware implementation has become an attractive alternative. Assigning complex computation tasks to hardware and exploiting the parallelism and pipelining in algorithms yield significant speedup in running times. In this thesis the image processing algorithms like median filter, basic morphological operators, convolution and edge detection algorithms are implemented on FPGA. A pipelined architecture of these algorithms is presented.

CHAPTER 1

INTRODUCTION

The field of image processing has grown enormously during the past decade. People working in an increasing range of disciplines and application areas are using image processing technology. The importance of the topic can be seen from the large number of papers presented at major national and international conferences. These cover many application areas, including defense, entertainment, security systems, personal identification, medical imaging, telecommunications, multimedia and others.

Generally speaking, low-level image processing operations involve performing computationally intensive but regular tasks on a large set of image data. So image processing application developers require high performance to speed up image processing applications. Over the years, extensive research work has been carried out to develop high performance image processing systems using various hardware approaches. The main approaches will first be reviewed in the following sections. Since this project will be based on using Xilinx Spartan II FPGA as the selected hardware give overviews of the Xilinx FPGA and other alternatives.

1.1 High Performance Solutions for Image Processing Applications

The relentless improvement in performance of mainstream microprocessors (e.g. 500Mhz Pentium III with MMX technology) means that it is now feasible to carry out quite a number of image processing applications using standard hardware. However, the research described in this thesis is concentrated on exploiting alternative hardware solutions for high performance image processing. In this section, we consider briefly three such methods: *parallel processing*, *Digital Signal Processing (DSP)* processors and *special purpose hardware*.

Parallel Processing

Like many other computationally intensive problems, parallel processing has been suggested as a possible solution for high

performance image processing. Most solutions have tended to be

based upon pure SIMD (Single-Instruction-Multiple-Data) or
more commonly, SPMD (Single-Program-Multiple-Data) type architectures. The typical SPMD approach involves distributing the image over a set of Processing Elements (PEs), with all of these PEs processing its own section of the image in parallel.
However, despite the huge amount of research and the advances in parallel processing over the past decade, the field of parallel processing is not yet mature. Programming a parallel machine is still considered difficult and the actual performance achieved on parallel computers is often only a fraction of their theoretical peak performance

Digital Signal Processing (DSP) processors

Perhaps the most common method of hardware accelerator for image processing and machine vision systems has been DSP-based (Digital Signal Processing) image processing boards. These products provide the computing power necessary to process large amounts of data in real-time. A DSP processor is tailored to perform repeatedly specific operations (such as MAC - multiplication followed by accumulation) very quickly. An advantage of the DSP approach is the more convenient
programming model (provided a good optimizing compilers and libraries are available).

Special purpose ASIC hardware

Architectural support in the form of special purpose ASIC or VLSI hardware can provide solutions that are specially tailored to the image-processing algorithm in hand. This approach has been used in the past because of the very high performance, which it can yield. Also for large production volume, cost can be lower than, say, a parallel processing.

There are some disadvantages to this approach however, which can be summarized as follow:

- Special purpose hardware has a long development time, from design through to simulation and fabrication.
- It can also be expensive if it is a one-off solution or if the volume required cannot justify its fabrication costs.
-

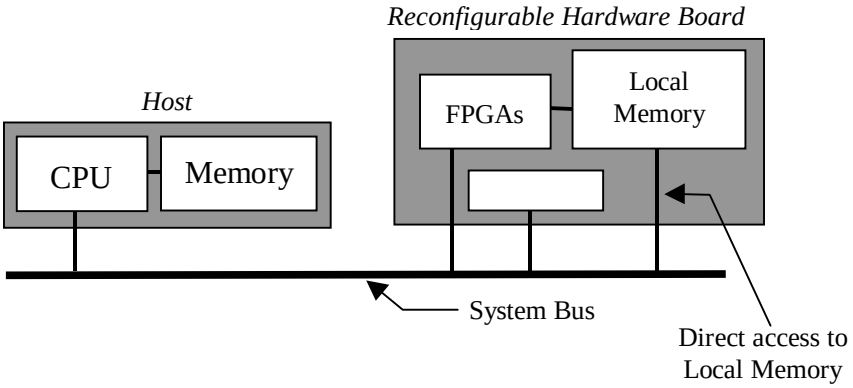
- Once this special purpose hardware is built, it is not possible to change the hardware to accommodate slightly different needs. With such a solution a new piece of hardware is usually required for each new algorithm.

A more recent approach, which aims to benefit from the advantages of special purpose.
hardware which avoiding many of its disadvantages, is to use dynamically reprogram able hardware in the form of Field Programmable Gate Arrays (FPGAs). This technology is now overviewed.

1.2 Reconfigurable Hardware – FPGAs

Since the mid 1980's, *reconfigurable hardware* devices in the form of Field Programmable Gate Arrays (FPGAs) have benefited from the same advances in IC fabrication technology as microprocessors. These devices aim to combine the flexibility of a programmable device (such as a general-purpose processor) with the performance of application specific special purpose hardware (such as ASICs).

Reconfigurable hardware presents an alternative technology that can be reconfigured as an application executes. With its inherent speed and adaptability, it seems an ideal candidate for image processing applications.A typical simple arrangement is for an FPGA to act as part of a coprocessor, which communicates with
a host processor via a bus interface and probably have shared memory (see **figure 1.1**)



In the past, most reconfigurable hardware boards have been realised as external boards attached to the system bus via an additional interface. But as FPGAs get denser, modern

reconfigurable hardware boards are realised as smaller extension boards, which can be plugged directly into the system bus.

FPGAs have traditionally been configured by hardware engineers using a Hardware *Design Language (HDL). The two principal languages being used are Verilog and VHDL.* Verilog and VHDL are specialized design techniques that are not

immediately accessible to software engineers, who have often been trained using imperative programming languages.

1.3 Motivation and Aims of the Thesis

The previous sections have given reasons why it is interesting and worthwhile to investigate further the exploitation of FPGAs for image processing applications.

For more detail, the disadvantages of using FPGAs identified previously improve the image processing application developer in several ways:

- The developer has to think very much in terms of hardware architectures rather than image processing algorithms.
- The developer must be able to use some form of hardware description language. The lower the level of the HDL, the more detail the developer must master.
- The design cycle, with its numerous intermediate stages, can be much slower than the traditional edit-compile-execute software cycle.

These unfortunate consequences are the underlying motivation for this thesis. The general goal is to support the image processing application developer in exploiting FPGAs by providing more appropriate software tools. In particular, the main aims are:

- To provide a high level-programming environment, this will help to bridge the gap between algorithms and architecture descriptions.
- To hide as much as possible of the details of the FPGA hardware and its environment. This in turn will reduce the learning curve.
- To speed up the design cycle by eliminating some of the intermediate stages, which are currently necessary. This is useful for rapid experimentation, which is important for developing image-processing applications.

1.4 Organization of the report

The organization of the rest of the thesis as follows

Organization of Report

- Chapter two provides information on FPGA's, Spartan II family since I have implemented on Spartan 2 (XC2S200PQ208).
- Chapter three describes the image processing algorithms like Median Filter, convolution and edge detection.
- Chapter four provides the details on the implementation of the image processing algorithms on a Xilinx Spartan 2
  FPGA for a320 x 240 gray scale image.
- Chapter five summaries the results and future work.

Chapter 2

Literature Review

FPGAs for Image Processing

With the emergence of reconfigurable hardware in the form of FPGAs, it is not surprising that there has been a considerable amount of research into the use of FPGAs to increase the performance of a wide range of computationally intensive applications. One such computationally intensive application that could greatly benefit from the advantages offered by FPGAs is *image processing*. The regular nature of the complex computations performed repeatedly within an image processing operation is well suited to a hardware based implementation using FPGAs.

In this background chapter, firstly gives a small number of examples to illustrate the current use of FPGAs for image processing applications. The chapter then moves on to consider Spartan II FPGA, the general description, features, architectural details, and finally various configuration modes in which FPGA can configured for image processing . Finally, the main objectives of this research are summarised at the end of the chapter.

2.1 FPGA-based Image Processing Applications

This section illustrates the range of uses of FPGAs for imageprocessing applications by selecting and presenting five different examples.

(i) Inspection Systems

AS&E (American Science and Engineering) introduced a set of new systems based on Univision technology which is a combination of a SVGA display, an image processor and a frame grabber. The image processor is based on FPGAs from Xilinx. These systems have been used for the automatic inspection of baggage, mail, cargo, people and vehicles.

AS&E submitted function definitions to Univision which were programmed into the Xilinx FPGA by Univision engineers. These functions include algorithms for image and edge enhancement, and a zoom function. Other features, such as pan, scroll and density expand are carried out in Univision's Falcon software. The image enhancement filtering specified by AS&E uses a 7x7 convolution, and is done using the FPGA.

(ii) Image Compression Using FPGAs

There are different examples of the use of FPGAs to speed up Video compression. The 2D Discrete Cosine Transform (2D DCT) is one of the most effective methods in image data compression. FPGAs are exploited for DCT implementation as mentioned in [Chu99] presents how to implement a block-matching motion algorithm efficiently by FPGAs for video compression. Segmentation algorithms which are sometimes used in image compression to obtain an image representation at different resolution levels. [Was99] shows an implementation of some segmentation algorithms on a single FPGA chip.

(iii) Medical Image Enhancement Using FPGA Technology

Features of medical images include:

- Images are subject to noise due to the limitations placed on the method of capture (e.g. X-ray doses).
- Images can lack resolution due to the fact that rays must pass through tissues of varying density (e.g. in MRI and Ultrasound scans).

Data rates in the medical imaging systems can also present the system designer with a formidable task. Typical systems have 8 or 16-bit pixels, a 512 x 512bit to 2048 x 2048bit image size, and 7.5 to 30 frames per second. Designing a system to enhance images in real time at these rates requires optimal kernel size to control costs. Therefore FPGA technology is being exploited to perform image processing tasks which require a large amount of computation.

FPGAs for medical image enhancement have been used in [RDTC98]. An enhancement algorithm has been implemented using a 15 x 15 image filter on a single chip (Xilinx XC4000 series). The enhancement algorithm is partitioned into a low pass filter and image mixing cores. This work forms the basis for image resizing applications, and noise reduction image filters.

As an another example of exploiting FPGAs for medical images, rendering volumetric medical images is a burdensome computational task for contemporary computers due to the large size of the data sets. [SAHL98] presents an algorithm and

speedup techniques for visualising volumetric medical MRI images with FPGAs.

(iv) Real Time Image Rotation

Based on applying B-spline interpolation functions, [Ber98] realised an implementation of a real-time high-quality rotation on Xilinx XC6200 series FPGAs. The rotation
algorithm is very computationally intensive and mathematically based. The FPGA architecture is very specific to the algorithm.

(v) Accelerating Adobe PhotoShop

Adobe PhotoShop is a widely used image processing package, which provides a modular architecture for extending its functionality based on plug-ins. PhotoShop provides filters that can manipulate an image (in true-colour 24-bit) in various ways including colour manipulation and filtering (e.g. Gaussian blur). For large images these filters can take a long time to run.

Brief Description Of Spartan-II 2.5V FPGA Family

Since I am going to implement this project on the Spartan II FPGA Let us have a look at the details of architecture of the device.

2.2 Introduction

The Spartan™-II 2.5V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The six-member family offers densities ranging from 15,000 to200,000 system gates, as shown in **Table1**. System performance is supported up to 200 MHz. Spartan-II devices deliver more gates, I/Os, and features per dollar than other FPGAs by combining advanced process technology with a streamlined Virtex-based architecture. Features include block RAM (to 56K bits), distributed RAM (to 75,264 bits), 16 selectable I/O standards, and four DLLs. Fast, predictable interconnect means that successive design iterations continue to meet timing requirements. The Spartan-II family is a superior alternative to mask-programmed ASICs. The FPGA avoids the initial cost, lengthy development cycles, and inherent risk of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary (impossible with ASICs).

2.3 Features

- Second generation ASIC replacement technology.
    - Densities as high as 5,292 logic cells with up to 200,000 system gates.
    - Streamlined features based on Virtex architecture.
    - Unlimited reprogrammability.
    - Very low cost.
- System level features
    - SelectRAM+™ hierarchical memory.
- 16 bits/LUT distributed RAM.
- Configurable 4K bit block RAM.
- Fast interfaces to external RAM.
    - Fully PCI compliant.
    - Low-power segmented routing architecture.
    - Full readback ability for verification/observability.
    - Dedicated carry logic for high-speed arithmetic.
    - Efficient multiplier support.
    - Cascade chain for wide-input functions.
    - Abundant registers/latches with enable, set, reset.
    - Four dedicated DLLs for advanced clock control.
    - Four primary low-skew global clock distribution nets.
    - IEEE 1149.1 compatible boundary scan logic.
- Versatile I/O and packaging.
    - Pb-free package options.
    - Low-cost packages available in all densities.
    - Family footprint compatibility in common pac.
    - 16 high-performance interface standards.
    - Hot swap Compact PCI friendly.
    - Zero hold time simplifies system timing.
- Fully supported by powerful Xilinx development system.
    - Foundation ISE Series: Fully integrated software.
- Alliance Series: For use with third-party tools.
    - Fully automatic mapping, placement, and routing.

2.4 General Overview

The Spartan-II family of FPGAs have a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. These functional elements are interconnected by a powerful hierarchy of versatile routing channels (see **Figure 2.4**).

Table 1

Device	Logic Cells	System gates (Logic and RAM)	CLB array R x C	Total CLBs	Maximum allowable user I/Os	Total Distributed RAM bits	Total Block RAM bits
XC2S 15	432	15,000	8 x 12	96	86	6,144	16K
XC2S 30	972	30,000	12 x18	216	92	13,824	24K
XC2S 50	1728	50,000	16 x24	384	176	24,576	32K
XC2S 100	2,700	100,000	20 x30	600	176	38,400	40K
XC2S 150	3,888	150,000	24 x36	864	260	55,296	48K
XC2S 200	5,292	200,000	28 x42	1,176	284	75,264	56K

Spartan-II FPGAs are customized by loading configuration data into internal static memory cells. Unlimited

reprogramming cycles are possible with this approach. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan modes.

Spartan-II FPGAs are typically used in high-volume applications where the versatility of a fast programmable solution adds benefits. Spartan-II FPGAs are ideal for shortening product development cycles while offering a cost-effective solution for high volume production.

Spartan-II FPGAs achieve high-performance, low-cost operation through advanced architecture and semiconductor technology. Spartan-II devices provide offer the most cost-effective solution while maintaining leading edge performance. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-II FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and reset on all flip-flops, fast carry logic, and many other distributed form), DLL clock drivers, programmable set and system clock rates up to 200 MHz. Spartan-II FPGAs features.

2.5 Spartan-II Product Availability

Table 2 shows the maximum user I/Os available on the device and the number of user I/Os available for each device/package combination. The four global clock pins are usable as additional

user I/Os when not used as a global c lock pin. These pins are not included in user I/O counts.

2.5.1 Ordering Information

Spartan-II devices are available in both standard and Pb-free packaging options for all device/package combinations. The Pb-free packages include a special "G" character in the ordering code.

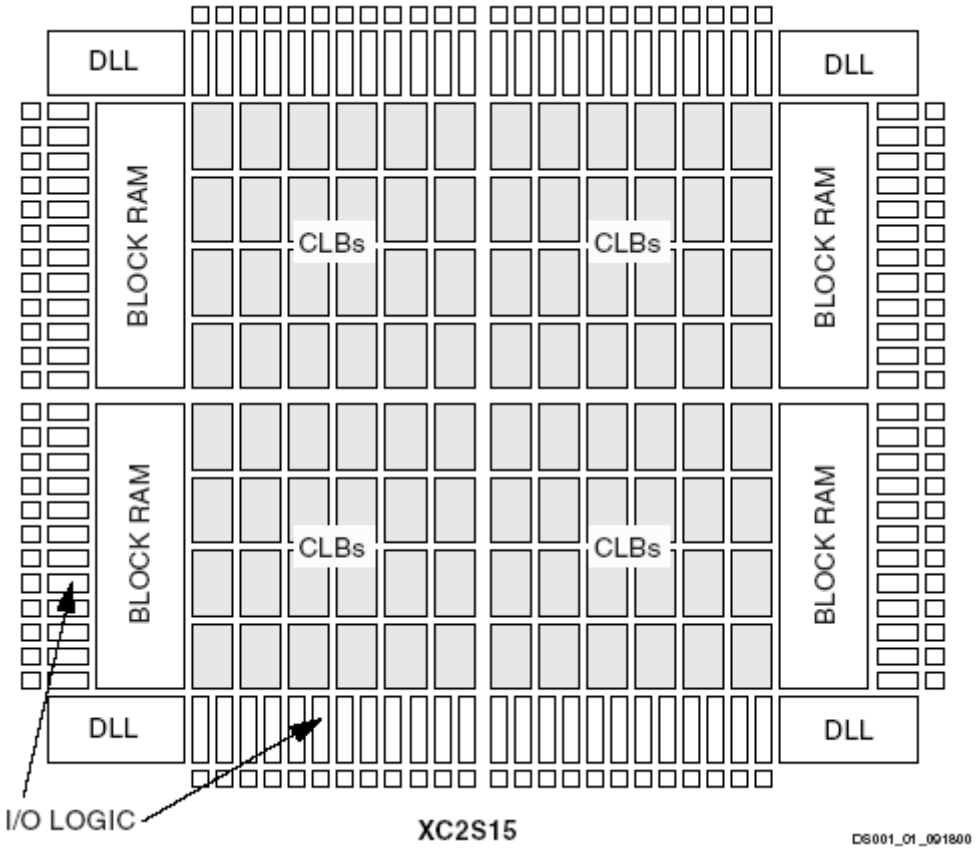
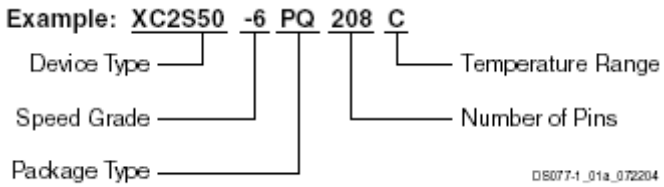


Figure 2.4 Basic Spartan II FPGA Family Block Diagram

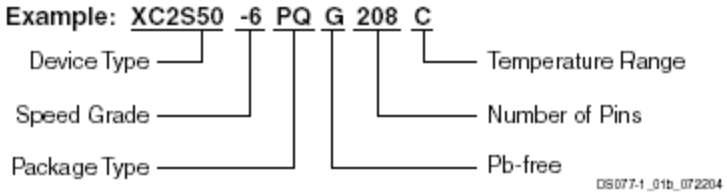
Table 2

Device	**Maximum user I/O**	**VQ100 VQG100**	**TQ144 TQG144**	**CS144 CSG144**	**PQ208 PQG208**	
XC2S15	86	60	86	-	-	
XC2S30	92	60	92	92	-	
XC2S50	176	-	92	-	140	
XC2S100	176	-	92	-	140	
XC2S150	260	-	-	-	140	
XC2S200	284	-	-	-	140	

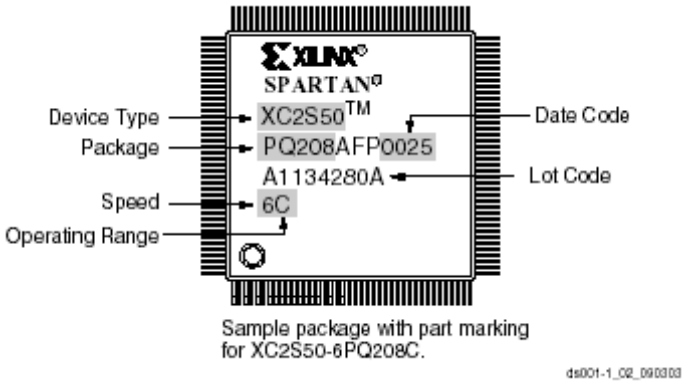
Standard Packaging



Pb-Free Packaging



Device Part Marking



2.6 Architectural Description

2.6.1 Spartan-II Array

The Spartan-II user-programmable gate array, shown in Figure 1, is composed of five major configurable elements :

• IOBs provide the interface between the package pins and the internal logic

• CLBs provide the functional elements for constructing most logic

• Dedicated block RAM memories of 4096 bits each

• Clock DLLs for clock-distribution delay compensation and clock domain control

• Versatile multi-level interconnect structure

As can be seen in Figure 1, the CLBs form the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip.
Values stored in static memory cells control all the configurable logic elements and interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device. Each of these elements will be discussed in detail in the following sections.

2.6.2 Input/Output Block

The Spartan-II IOB, as seen in Figure 1, features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state of the art memory and bus interfaces. **Table 1** lists several of the standards which are supported along with the required reference, output and termination voltages needed to meet the standard.
The three IOB registers function either as edge-triggered D-type flip-flops or as level-sensitive latches. Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable (CE) signals for each register.

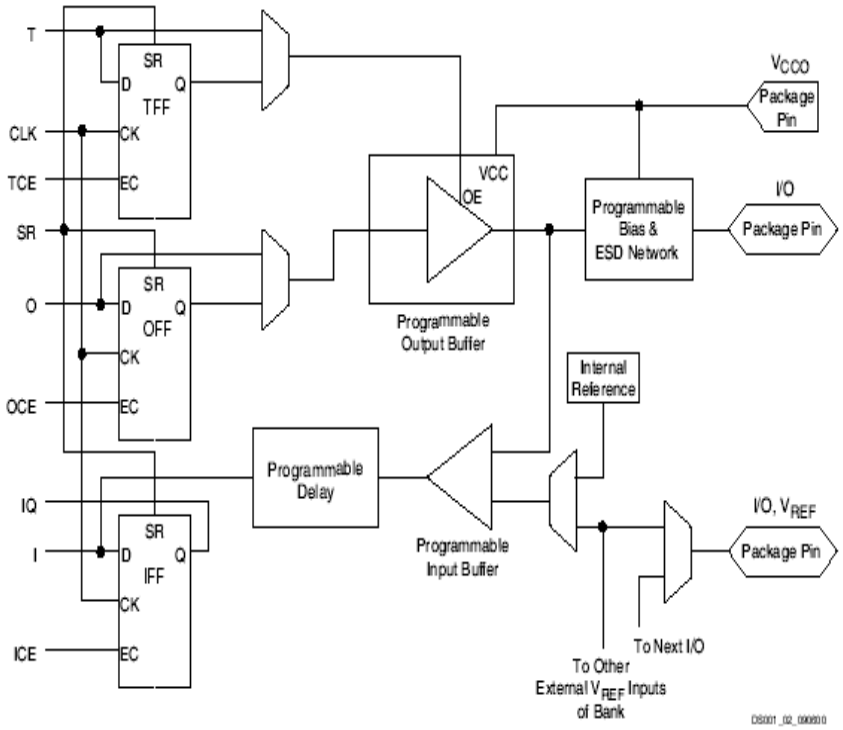


Figure 2.6 Spartan 2 INPUT/OUTPUT BLOCK (IOB)

Input Block

In addition to the CLK and CE control signals, the three reg isters share a Set/Reset (SR). For each register, this signalcan be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear.

A feature not shown in the block diagram, but controlled by the software, is polarity control. The input and output buffers and all of the IOB control signals have independent polarity controls. Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each pad. Prior to configuration all outputs not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive, but inputs may optionally be pulled up.

Table 3: Standards Suppor ted by I/O (Typical Values)

IO Standared	**Input Reference Voltage (V_{ref})**	**Output Source Voltage (V_{cco})**	Board Termination Voltage **(V_{TT})**
LVTTL(2-24 mA)	N/A	3.3	N/A

LVCMOS2	N/A	2.5	N/A
PCI(3V/5V, 33MHz/66MHz)	N/A	3.3	N/A
GTL	0.8	N/A	1.2
GTL+	1.0	N/A	1.5
HSTL Class I	0.75	1.5	0.75
HSTL Class III	0.9	1.5	1.5
HSTL Class IV	0.9	1.5	1.5
SSTL3 Class I and II	1.5	3.3	1.5
SSTL2 Class I and II	1.25	1.5	1.5
CTT	1.5	3.3	1.5
AGP-2X	1.32	3.3	N/A

Input Path

The activation of pull-up resistors prior to configuration is controlled on a global basis by the configuration mode pins. If the pull-up resistors are not activated, all the pins will float. Consequently, external pull-up or pull-down resistors must be provided on pins required to be at a well-defined logic level prior to configuration. All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. Two forms of over-voltage protection are provided, one that permits 5V compliance, and one that does not. For 5V compliance, a zener-like structure connected to ground turns on when the output rises to approximately 6.5V. When 5V compliance is not required, a conventional clamp diode may connected to the output supply voltage, VCCO. The type of over-voltage protection can be selected independently for each pad.

All Spartan-II IOBs s support IEEE 1149.1-compatible boundary scan testing.

2.6.3 Input Path

A buffer In the Spartan-II IOB input path routes the input signal either directly to internal logic or through an optional input flip-flop. An optional delay element at the D-input of this flip-flop eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of

the FPGA, and when used, assures that the pad-to-pad hold time is zero. Each input buffer can be configured to conform to any of the low-voltage Signaling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, VREF. The need to supply VREF imposes constraints on which tandards can used in close proximity Voltage Volt age ge to each other. See I/O Banking,. I/O Standard (VREF) (VCCO) (VTT) There are optional pull-up and pull-down resistors at each input for use after configuration.

2.6.4 Output Path

The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to
the buffer directly from the internal logic or through an optional IOB output flip-flop. The 3-state control of the output can also be routed directly from the internal logic or through a flip-flip that provides synchronous enable and disable.

Output Path

Each output driver can be individually programmed for a wide range of low-voltage signaling standards. Each output buffer can source up to 24 mA and sink up to 48 mA. Drive strength and slew rate controls minimize bus transients In most signaling standards, the output high voltage depends on an externally supplied VCCO voltage. The need to supply VCCO imposes constraints on which standards can be used in close proximity to each other. See I/O Banking**.**An optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low to match the input signal. If the pin is connected to a multiple-source signal, the weak keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way helps eliminate bus chatter. Because the weak-keeper circuit us es the IOB input buffer to monitor the input level, an appropriate VREF voltage must be provided if the signaling standard requires one. The provision of this voltage must comply with the I/O banking rules.

2.6.5 I/O Banking

Some of the I/O standards described above require VCCO and/or VREF voltages. These voltages are externally connected to device pins that serve groups of IOBs, called banks. Consequently,

restrictions exist about which I/O standards can be combined within a given bank. Eight I/O banks result from separating each edge of the FPGA into two banks (see **Figure 2.6**). Each bank has multiple VCCO pins which must be connected to the same voltage. Voltage is determined by the output standards in use.

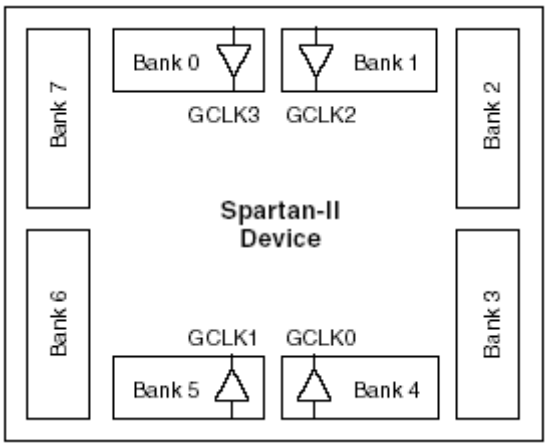


Figure 2.6. spartan 2 IO Banks

Output Path

Within a bank, output standards may be mixed only if they use the same VCCO. Compatible standards are shown in **Table 4**. GTL and GTL+ appear under all voltages because their open-drain outputs do not depend on VCCO.

Some input standards require a user-supplied threshold voltage, VREF. In this case, certain user-I/O pins are automatically configured as inputs for the VREF voltage. About one in six of the I/O pins in the bank assume this role. VREF pins within a bank are interconnected internally and consequently only one VREF voltage can be used within each bank. All VREF pins in the bank, however, must be connected to the external voltage source for correct operation. In a bank, inputs requiring VREF can be mixed with those that do not but only one VREF voltage may be used within a bank. Input buffers that use VREF are not 5V tolerant. LVTTL, LVCMOS2, and PCI are 5V tolerant. The VCCO and VREF pins for each bank appear in the device pinout tables. Within a given package, the number of VREF and VCCO pins can vary depending on the size of device. In larger devices, more I/O pins convert to VREF pins. Since these are always a superset of the VREF pins used for smaller devices, it is possible to design a PCB that

permits migration to a larger device. All VREF pins for the largest device anticipated must be connected to the VREF voltage, and not used for I/O.

V_{cco}	Compatible standareds
3.3V	PCI, LVTTL, SSTL3 I, SSTL3 II, CTT, AGP, GTL, GTL+
2.5V	SSTL2 I, SSTL2 II, LVCMOS2, GTL, GTL+
1.5V	HSTL 1, HSTL3 III, HSTL IV, GTL, GTL+

Table 4 Compatible output standards

2.7 Configurable Logic Block

The basic building block of the Spartan-II CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and storage element. Output from the function generator in each LC drives the CLB output and the D input of the flip-flop. Each Spartan-II CLB contains four LCs, organized in two similar slices; a single slice is shown in **Figure**

Configurable Logic Block

2.7.In addition to the four basic LCs, the Spartan-II CLB contains logic that combines function generators to provide functions of five or six inputs.

2.7.1 Look-Up Tables

Spartan-II function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT c an provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a
slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16 x 1-bit dual-port synchronous RAM. The Spartan-II LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in applications such as Digital Signal Processing.

2.7.2 Storage Elements

Storage elements in the Spartan-II slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by function generators within the slice or directly from slice inputs, bypassing the function generators.
In addition to Clock and Clock Enable signals, each slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialization state specified.
for it in the configuration. BY forces it into the opposite state. Alternatively, these signals may be configured to operate asynchronously. All control signals are independently invertible, and are shared by the two flip-flops within the slice.

2.7.3 Additional Logic

The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine inputs.
Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs.Each CLB has

Storage Elements

---

four direct feed through paths, one per LC. These paths provide extra data input lines or additional local routing that does not consume logic resources.

2.7.4 Arithmetic Logic

Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Spartan-II CLB supports two separate carry chains, one per slice. The height of the carry chains is two bits per CLB

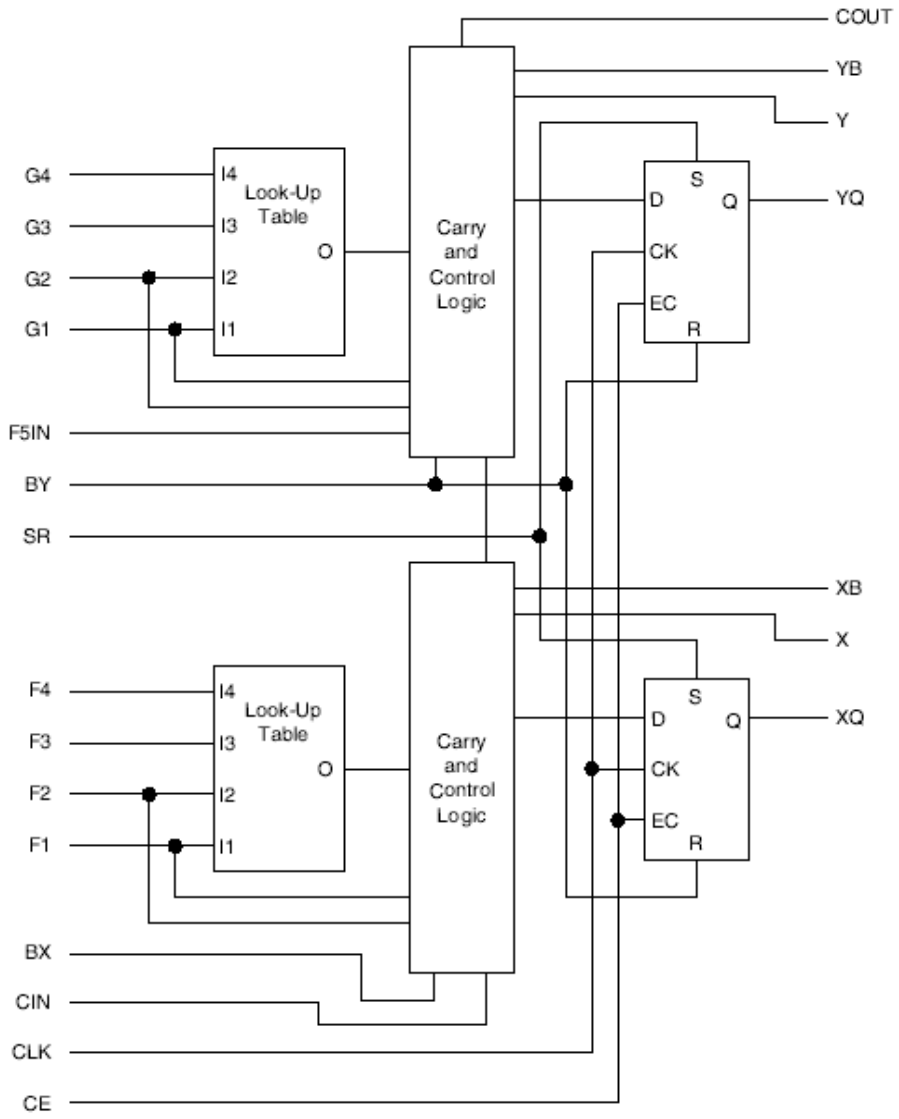


Figure 2.7 Spartan II CLB Slice (two identical slices in each CLB)

Programmable Routing Matrix

The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementation. The dedicated carry path can also be used to cascade function generators for implementing wide logic functions.

2.8 Programmable Routing Matrix

It is the longest delay path that limits the speed of any worst-case design. Consequently, the Spartan-II routing architecture and its place-and-route software were defined in a single

optimization process. This joint optimization minimizes long-path delays, and consequently, yields the best system performance. The joint optimization also reduces design compilation times because the architecture is software-friendly. Design cycles are correspondingly reduced due to shorter design iteration times.

2.8.1 Local Routing

The local routing resources, as shown in **Figure 2.8**, provide the following three types of connections

- Interconnections among the LUTs, flip-flops, and General Routing Matrix (GRM)
- Internal CLB feedback paths that provide high-speed connections to LUTs within the same CLB, chaining.them together with minimal routing delay
- Direct paths that provide high-speed connections between horizontally adjacent CLBs, eliminating the delay of the GRM .

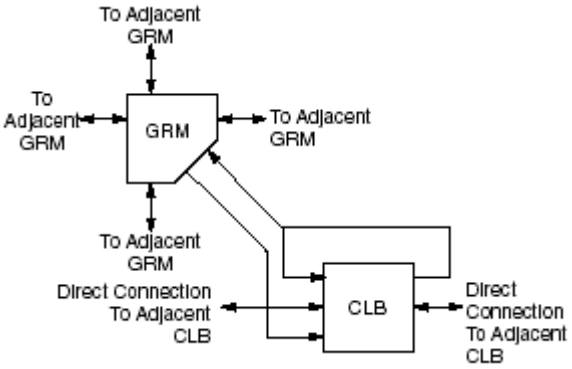


Figure 2.8 Spartan II Local Routing

General Purpose Routing

2.8.2 General Purpose Routing

Most Spartan-II signals are routed on the general purpose routing, and consequently, the majority of interconnect resources are associated with this level of the routing hierarchy. The general routing resources are located in horizontal and vertical routing channels associated with the rows and columns CLBs. The general-purpose routing resources are listed below

- Adjacent to each CLB is a General Routing Matrix (GRM). The GRM is the switch matrix through which
- horizontal and vertical routing resources connect, and is also the means by which the CLB gains access to the general purpose routing.
- 24 single-length lines route GRM signals to adjacent GRMs in each of the four directions.
- 96 buffered Hex lines route GRM signals to other GRMs six blocks away in each one of the four directions. Organized in a staggered pattern, Hex lines may be driven only at their endpoints. Hex-line signals can be accessed either at the endpoints or at the midpoint (three blocks from the source). One third of GRM the Hex lines are bidirectional, while the remaining ones are unidirectional.
- 12 Long lines are buffered, bidirectional wires that GRM distribute signals across the device quickly and
- Efficiently. Vertical Long lines span the full height of the device, and horizontal ones span the full width of the device.

2.8.3 I/O Routing

Spartan-II devices have additional routing resources around their periphery that form an interface between the CLB array and the IOBs. This additional routing, called the Versa Ring, facilitates pin-swapping and pin locking, such that logic redesigns can adapt to existing PCB layouts. Time-to-market is reduced, since PCBs and other s ystem components can be manufactured while the logic design is still in progress.

2.8.4 Dedicated Routing

Dedicated Routing

Some classes of signal require dedicated routing resources to maximize performance. In the Spartan-II architecture, dedicated routing resources are provided for two classes of signal.

Horizontal routing resources are provided for on-chip 3-state busses. Four partition able bus lines are provided per CLB row, permitting multiple busses within a row, as shown in **Figure 2.8.4.**Two dedicated nets per CLB propagate carry signals vertically to the adjacent CLB.

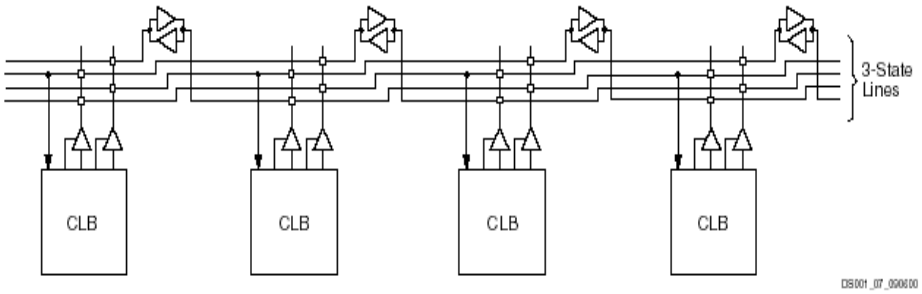


Figure 2.8.4 BUFT connection to Dedicated Horizontal Bus Lines

2.8.5 Global Routing

Global Routing resources distribute clocks and other signals with very high fan-out throughout the device. Spartan-II devices include two tiers of global routing resources referred to as primary and secondary global routing resources.

- The primary global routing resources are four dedicated global nets with dedicated input pins that are designed to distribute high-fan-out clock signals
- with minimal skew. Each global clock net can drive all CLB, IOB, and block RAM clock pins. The primary global nets may only be driven by global buffers. There are four global buffers, one for each global net.
- The secondary global routing resources consist of 24 backbone lines, 12 across the top of the chip and 12
- Across bottom. From these lines, up to 12 unique signals per column can be distributed via the 12 loglines in the column. These secondary resources are more flexible than the primary resources since they are not restricted to routing only to clock pins.

2.9 Clock Distribution

Clock Distribution

The Spartan-II family provides high-speed, low-skew clock distribution through the primary global routing resources described above. A typical clock distribution net is shown in **Figure 2.9**. Four global buffers are provided, two at the top center of the device and two at the bottom center. These drive the four primary global nets that in turn drive any clock pin.

Four dedicated clock pads are provided, one adjacent to each of the global buffers. The input to the global buffer is selected either from these pads or from signals in the general

purpose routing. Global clock pins do not have the option for internal, weak pull-up resistors.

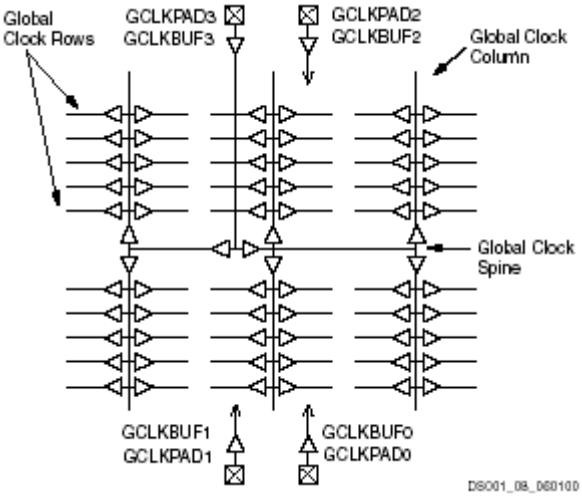


Figure 2.9 Global Clock Distribution Network

2.10 Delay-Locked Loop (DLL*)*

Associated with each global clock input buffer is a fully digital Delay-Locked Loop (DLL) that can eliminate skew between the clock input pad and internal clock-input pins throughout the device. Each DLL can drive two global clock networks. The DLL monitors the input clock and the distributed clock, and automatically adjusts a clock delay element. Additional delay is introduced such that clock edges reach internal flip-flops exactly one clock period after they arrive at the input. This closed-loop system effectively eliminates clock-distribution delay by ensuring that clock edges arrive at internal flip-flops in synchronism with clock edges arriving at the input.

In addition to eliminating clock-distribution delay, the DLL provides advanced control of multiple clock domains. The DLL provides four quadrater phases of the source clock,

can double the clock, or divide the clock by 1.5, 2, 2.5, 3, 4, 5, 8, or 16. It has six outputs. The DLL also operates as a clock mirror. By driving the output from a DLL off-

Delay Locked Loop(DLL)

---

chip and then back on again, the DLL can be used to deskew a board level clock among multiple Spartan-II devices.

In order to guarantee that the system clock is operating correctly prior to the FPGA starting up after configuration, the DLL can delay the completion of the configuration process until after it has achieved lock.

2.11 Design Implementation

The place-and-route tools (PAR) automatically provide the implementation flow described in this section. The practitioner takes the EDIF net list for the design and maps the logic into the architectural resources of the FPGA (CLBs and IOBs, for example). The placer then determines the best locations for these blocks based on their interconnections and the desired performance. Finally, the router interconnects the blocks.

The PAR algorithms support fully automatic implementation of most designs. For demanding applications, however, the user can exercise various degrees of control over the process. User partitioning, placement, and routing information is optionally specified during the design-entry process. The implementation of highly structured designs can benefit greatly from basic floor planning.

The implementation software incorporates Timing Wizard® Timing-driven placement and routing. Designers specify timing requirements along entire paths during design

entry. The timing path analysis routines in PAR then recognize these user-specified requirements and accommodate them.

Timing requirements are entered on a schematic in a form directly relating to the system requirements, such as the targeted clock frequency, or the maximum allowable delay between two registers. In this way, the overall performance of the system along entire signal paths is automatically tailored to user-generated specifications. Specific timing information for individual nets is unnecessary.

2.12 Design Verification

In addition to conventional software simulation, FPGA users can use in-circuit debugging techniques. Because Xilinx devices are infinitely reprogrammable, designs can be verified in real time without the need for extensive sets of software simulation vectors.

Configuration

The development system supports both software simulation and in-circuit debugging techniques. For simulation, the system extracts the post-layout timing information from the design database, and back-annotates this information into the net list for use by the simulator. Alternatively, the user can verify timing-critical portions of the design using the static timing analyzer.

For in-circuit debugging, the development system includes a download and read back cable, which connects the FPGA in the target system to a PC or workstation. After downloading the design into the FPGA, the designer can single-step the logic, read back the contents of the flip-flops, and so observe the internal logic state. Simple modifications can be downloaded into the system in a matter of minutes.

2.13 Configuration

Configuration is the process by which the bits stream of a design, as generated by the Xilinx development software, is loaded into the internal configuration memory of the FPGA. Spartan-II devices support both serial configurations, using the master/slave serial and JTAG modes, as well as byte-wide configuration employing the Slave Parallel mode.

2.13.1 Modes

Spartan-II devices support the following four configuration modes:

- Slave Serial mode.
- Master Serial mode.
- Slave Parallel mode.
- Boundary-scan mode.

The Configuration mode pins (M2, M1, M0) s elect among these configuration modes with the option in each case of having the IOB pins either pulled up or left floating prior to configuration. Configuration through the boundary-scan port is always available, independent of the mode selection. Selecting the boundary-scan mode simply turns off the other modes. The
three mode pins have internal pull-up resistors, and default to a logic High if left unconnected.

Configuration

2.13.1.1Serial Modes

There are two serial configuration modes: In Master Serial mode, the FPGA controls the configuration process by driving CCLK as an output. In Slave Serial mode, the FPGA

passively receives CCLK as an input from an external agent (e.g., a microprocessor, CPLD, or second FPGA in master mode) that is controlling the configuration process. In both modes, the FPGA is configured by loading one bit per CCLK
cycle. The MSB of each configuration data byte is always written to the DIN pin first.

2.13.1.2 Slave Parallel Mode

The Slave Parallel mode is the fastest configuration option. Byte-wide data is written into the FPGA. A BUSY flag is provided for controlling the flow of data at a clock frequency FCCNH above 50 MHz.

2.13.1.3 Boundary-Scan Mode

In the boundary-scan mode, no nondedicated pins are required, configuration being done entirely through the IEEE 1149.1 Test Access Port.
Configuration through the TAP uses the special CFG_IN instruction. This instruction allows data input on TDI to be converted into data packets for the internal configuration bus.
The following steps are required to configure the FPGA through the boundary-scan port.

1. Load the CFG_IN instruction into the boundary-scan instruction register (IR)
2. Enter the Shift-DR (SDR) state
3. Shift a standard configuration bitstream into TDI
4. Return to Run-Test-Idle (RTI)
5. Load the JSTART ins truction into IR
6. Enter the SDR state
7. Clock TCK through the sequence (the length is programmable)
8. Return to RTI.

Configuration and read back via the TAP is always available. The boundary-scan mode simply locks out the other modes. The boundary-scan mode is selected by a <10x> on the mode pins (M0, M1, M2).

Objective Of Proposed Project

2.13.1.4 Readback

The configuration data stored in the Spartan-II configuration memory can be readback for verification. Along with the configuration data it is possible to readback the contents of all flip-flops/latches, LUT RAMs, and block RAMs. This capability is used for real-time debugging. For more detailed information see XAPP176, Spartan-II FPGA Family Configuration and Readback**.**

2.14 Objectives of the Proposed Project

- It should enable the definition of ***scaleable*** and ***parameterisable*** architectures and building blocks (e.g. to allow experimentation with different word lengths, or use different template window sizes).
- It should be easy to use and easy to learn, and should not appear overly mathematical.
- It should enable ***first-time-right place and route***, but without requiring the architecture designer to be responsible for placement and routing. The environment should apply 'common sense' placement and routing rules.
- It should be ***convenient*** for image processing operations, which shields the user from using low level details of hardware description.
- The description notation should be closer to the application domain to bridge the gap between hardware and image processing application.
- The image processing application developer should be able to ***experiment*** with different image algebra-based operations.
- The ***design cycle*** should be a rapid and convenient as possible.

Chapter 3

System Review

3.1 Convolution

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

3.2 1 -D Convolution

The *convolution operation* is a mathematical operation which takes two functions *f*(*x*) and *g*(*x*) and produces a third function *h*(*x*). Mathematically, convolution is defined as:

$$h(x) = f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau) g(x-\tau) d\tau$$

g(*x*) is referred to as the *filter*.

3.3 2D-Convolution

2D-Convolution is most important to modern image processing. The basic idea is that a window of some finite size and shape is scanned over an image. The output pixel value is the weighted sum of the input pixels within the window where the weights are the values
of the filter assigned to every pixel of the window. The window with its weights is called the *convolution mask.* Mathematically, convolution on image can be represented by the

$$y(m,n) = \sum_{i=0}^{Heightof\ image} \sum_{j=0}^{widthof\ image} h(i,j)\ x(m-i,n-j),$$

following equation.

31image the is y and filter the is h image input the is x where

An important aspect of convolution algorithm is that it supports a virtually infinite variety of masks, each with its own feature. This flexibility allows many powerful applications. 3x3 convolution masks are most commonly used, For example the derivative operators, which are mostly, used in edge detection use 3x3 window kernels. They operate only a pixel and its directly adjacent neighbors. Figure 3.5 shows a 3x3 convolution mask operated on an image. The center pixel is replaced with the output of the algorithm; this is carried for the entire image. Similarly larger size convolution masks can be operated on an image.

The convolution basically consists of sum of (delayed) products; therefore multiplication is an essential operation. Consequently, each multiplier can be implemented separately, and then the addition applied. Nevertheless, disregarding the multiplier entities allows for further optimizations. For example, for the LM, instead of considering separately additions within the multipliers and then the final addition, a single adders block can be formed, which allows for better grouping the adders, and therefore for implementing a more hardware-efficient circuit. This design approach to the group of the LMs is further denoted as LUT based Convolution (LC).

In addition, a (parallel) Distributed Arithmetic Convoler (DAC) – a completely different architectural solution can be implemented. This solution is similar to the LM, nevertheless, the order of multiplications and additions is disregarded, which allows for memory data width reduction, in comparison to the LC. This chapter presents also a novel approach: an Irregular Distributed Arithmetic Convolver (IDAC) which is a combination of the DAC and LC.
Unlike for multiplier less multiplication (MM), for convolution common substructure is not considered separately within each multiplier, but substructure sharing is applied for all coefficient altogether. Therefore a term, Multiplier less Convolution (MC), instead of the MM, is introduced. This causes that trading-off between the (LUT based) IDAC versus the MC is more complex than it is the case for the multiplication and the LM vs. MM. Consequently a sophisticated algorithm has been developed to confront the problem.

For convolution interdependence between coefficients is often very strong, as symmetric filters are often implemented. Consequently, additional algorithm for automatic detecting and grouping similar coefficients is also implemented.

3.4. Previous Works

For ASICs, several FIR filter silicon compilers have been developed. Nevertheless, FPGA designs differ significantly from the ASICs, as FPGAs usually incorporate dedicated ripple-carry logic, which makes that the different adders approach is adopted. Furthermore, FPGAs implement logic employing Look-Up Tables and therefore LUT-based multiplication or convolution is an alternative solution to be taken into account.

An automatic implementation of FIR filters on FPGAs has been presented in .This tool employs inverted form 1D FIR filters and techniques adopted for ASICs, such as power-of-two coefficient space (denoted hereby as the multiplierless multiplication), carry-save adders for XC3000 family [Xil93] (XC3000 family does not incorporate dedicated ripple-carry logic) and dedicated ripple-carry adders for XC4000 . Up to the author's knowledge, the SS has not been implemented in , only CSD representation is used. Employing inverted form FIR filters

instead of direct form filters (implemented by the AuToCon) excludes implementation of distributed arithmetic. Furthermore, adders operate on wider arguments in comparison to the direct-form filters.Besides inverted form filters are not recommended for 2D filters as wider line buffers are required.

Nevertheless, a 2D filter can be constructed from several inverted form 1D filters. Conversely, the structure of inverted form filters is more modular. Furthermore, a lot of design effort in has been put into mapping (optimising placment and routing) to increment clock frequency. Pipelining is (somehow) built-in the structure of the inverted form filters, therefore, in comparison to the pipeline architecture of the direct-form filter, it might seem that less flip-flops are required. However inverted-form filters require wider pipelining registers. Summing up, direct-form filters allow more architectural solutions to be adopted and more design parameters to be specified in comparison to the

inverted-form filters. Therefore, the direct-form solution has been adopted in the AuToCon, nevertheless more thorough research is required to compare these two different architectural solutions. Besides direct form filters can be employed as a sum of products, etc.

Core Generator, program distributed by Xilinx Inc., automatically generates FIR filters employing only (parallel) distributed arithmetic. Nevertheless, implementation results obtained for the

AuToCon outperform the results obtained for Core Generator, as the AuToCon considers different architectural solutions and applies more sophisticated

optimisation techniques. The Core Generator takes into account mapping of element into the FPGA. Conversely, the AuToCon generates circuits on higher level using VHDL-approach, therefore it might seem that the throughput for the Core Generator circuit is greater.Implementation results proved that this is not the case.

Xilinx Inc. also provides a VHDL-based FIR filter description employing inverted-form and KCM approach . Nevertheless, the input width is fixed and only number of taps and coefficient values can be changed. Besides, the KCM LUTs operate on 4 times the input clock frequency, therefore comparison with the AuToCon is impossible. It should be noted that there is a tendency for FPGAs to describe systems on high level (e.g. VHDL) and disregard relative placement of elements. This allows for reducing design time and/or implementing more sophisticated optimisation techniques.

3.5 Symmetry of Convolution Coefficients

Values of coefficients, in general, can be selected without any restrictions, however filters with symmetry are usually implemented, e.g. to obtain linear phase filters The filters given in Figure 1-1 are also with symmetry (or asymmetry in the case

of Sobel gradient filter). Table 5-1 gives possible 3x3 convolution kernels for different symmetries.

a)

W0,0	W0,1	W0,2
W1,0	W1,1	W1,2
W2,0	W2,1	W2,2

c)

W0,0	W0,1	W0,2
W1,0	W1,1	W1,2
W0,0	W0,1	W0,2

b)

W0,0	W0,1	W0,0
W1,0	W1,1	W1,0
W2,0	W2,1	W2,0

d)

W0,0	W0,1	W0,0
W1,0	W1,1	W1,0
W0,0	W0,1	W0,0

e)

W0,0	W0,1	W0,0
W0,1	W1,1	W0,1
W0,0	W0,1	W0,0

Table *Different Filter Symmetries a) with out symmetry b) horizontal c) vertical d) horizontal-vertical e) point symmetry*

The symmetry of the filter allows further optimisation of the circuit. The same coefficient inputs should be at first added, and then the common multiplication performed. Figure 5-1 shows the circuit simplifications, and Table 5-2 number of adders and multipliers after symmetry has been taken into account. It can be seen that for horizontal and vertical symmetry the number of multipliers is the same. However the number of adders and pixel delay elements is reduced for vertical symmetry because for this symmetry, only a single adder is needed for every common line. For the point symmetry, the number of multipliers is further reduced. It should be noted that for 3 3 convolution kernels, savings are less significant than for large kernel sizes, for which the number of multipliers is halved for horizontal or vertical symmetry, quartered for horizontal-vertical symmetry and reduced to 1/8 for point symmetry. It should be noted that for some 2D filters with the point symmetry, it

may be beneficial to implement two independent vertical and horizontal 1D filtering. This is often the case for wavelet transforms.

3.6 LUT based Convoler (LC)

3.6.1. Concept

The structure of the LUT based Convoler (LC) is similar to the sum of products. owever to optimise the structure of the adders, all additions are performed within a single adders block, therefore multiplier entities are disregarded. To illustrate savings obtained by the use of the LC instead of the sum of the LMs, an example is given in Figure 5-2, for convolution kernel size equal 1 2 and 8 8 multipliers. Let consider savings obtained by disregarding the multiplier bounds, for LUT output width equal $w= 12$ and LUT address width (shift between the same multiplier LUTs) $s= 4$. For the LM, the adder width within a multiplier equals roughly w. The final adder width equals roughly $w+s$. Therefore total adders width for the sum of the LM is equal

$$wLM= 3 \quad w+s.$$

For the LC, three adders of width equal w are employed, and therefore total number of Full /Half Adders is equal

$$wLC= 3 \quad w.$$

Consequently, a penalty factor, a result of employing sum of LMs instead of the LC, is roughly

$$p = \frac{w_{LM} - w_{LC}}{w_{LC}} = \frac{s}{3 \cdot w}$$

The above penalty factor is further employed for substructure sharing adders when two arguments are shifted by s. It should be also noted that employing the LC rather than the sum of LMs reduces the maximum width of the adder from roughly $w+s$ to w, and therefore reduces maximum propagation time.

3.6.2. Constant coefficients LUT based Convoler (KLC)

The KLC employs the same optimisation techniques as the KCM: LSB Address Width Reduction (LAWR), Don’t Care Address Width Reduction (DAWR) and Memory Sharing (MS). In addition, optimisation techniques characteristic only for convolers are employed .

Similar Coefficients Optimisation (SCO)

Section 5.2 describes the symmetries of filters. However, also different symmetries and coefficient combinations can be used [Lu92]. Therefore, the AuToCon compares all coefficients and groups them into similar coefficients blocks. Coefficients grouped together can be shifted and negated. Grouped inputs are shifted in respect to the coefficient value and then added (subtracted). Finally, a single multiplier is only implemented. This method allows for reducing the number of multipliers. For example, for the filter:

$$H(z) = H_1(z) + 5 \cdot z^{-i} - 5 \cdot z^{-j} - 10 \cdot z^{-k} + 20 \cdot z^{-l}$$

similar coefficient inputs are added:

$$A_5 = z^{-i} - z^{-j} - 2 \cdot z^{-k} + 4 \cdot z^{-l},$$

and the final result is:

$$H(z) = H_1(z) + 5 \cdot A_5.$$

In this example the number of multipliers has been reduced by3.

Pipelining Optimisation

additional parameter *p* defines maximum number of
logic elements between pipelining registers. Figure 5-3a shows an example of a convoler with straightforward pipelining architecture. For this method, however, additional pipelining registers are often required to compensate different pipelining delays. To reduce this drawback, pipelining optimisation is implemented, for which feeding points of arithmetic units are relocated in order to reduce unnecessary registers (similar optimisation is implemented in [Har96]). A result of the . It should be noted that the total convoler pipelining delay is often reduced in this method. This optimisation technique is implemented for every architecture described in this chapter.

3.6.3. Dynamic Constant coefficients LUT based Convoler (DKLC)

For the DKLC, the value of coefficients can be changed in similar way, as it is in the case for the DKCM; rearranging the order of adders does not influence the LUTs programming schedule. For the DKCM, address multiplexing is performed on the input of the multiplier.

Similarly for the DKLC, the multiplexer can be placed on the input of each multiplier. Let denote this option as DKLC-M. An alternative solution, denoted as DKLC-C, is to place the multiplexer on the convoler input and so the address sequence for programming LUTs will propagate through the convolution delay elements to the input of the LUTs. The drawback of this method is more sophisticated control logic. Besides, the number of programming cycles increases because of additional propagation time through the filter delay elements. In order to reduce this time the multiplexers should be rather placed at the beginning of each line. Therefore the programming sequence will propagate only through pixel delay elements. Summing up, *M* ,*N* (*M*- horizontal; *N*- vertical kernel size) convoler requires *N* multiplexers and *M-1* additional programming cycles for dynamic reconfiguration. In this option, however, similar coefficient adders (for symmetric filters, etc.) distract memory addressing and make the approach more complicated.

It should be noted that the LUTs can be programmed either in serial: a single multiplier is programmed at the time, or in parallel, when all multipliers are programmed simultaneously. The serial option has longer programming time but a single RAM Programming Unit (RPU) is required. The parallel option has short programming time but each multiplier requires its own RPU and therefore this option occupies more hardware. The choice between the serial and parallel option should be taken after considering the average time between coefficients changes in similar way as it was described in Section 4.6.

It should be noted that so far only self-programming architecture of the DKCM and DKLC has been considered. However, the LUTs can be programmed using an off-chip interface. In this case new LUT contents can be pre-calculated by a system processor and then written to the LUT memories. In this case the RPU is not required. Conversely, off-chip transfers are slower than internal ones and involve the system processor, which may not be accepted in some designs.

The DKLC can be implement with many different options. This is one of the reasons that the AuToCon cannot generate automatically any DKLC. Consequently including the DKLC to the AuToCon might be a suggestion for further work. Nevertheless Virtex II
incorporates built-in fully functional multipliers, which makes the DKLC option less attractive.

Distributed Arithmetic Convolver

3.7 Distributed Arithmetic Convoler (DAC)

3.7.1. Concept

The idea behind the DAC [Bur77, Min92, Do98] is to compute the convolution in different order than for the LC. The following mathematical transformation has been employed:

$$\sum_{i=0}^{N-1} h_i \cdot a_i = \sum_{i=0}^{N-1} h_i \cdot \sum_{j=0}^{L-1} 2^j \cdot a_{i,j} = \sum_{j=0}^{L-1} 2^j \cdot \sum_{i=0}^{N-1} h_i \cdot a_{i,j}$$

where: N- size of the convolution kernel, L- width of the input argument a (in bits), hi- ith coefficient of the convolution, ai,j- -j-th bit of the i-th input argument.

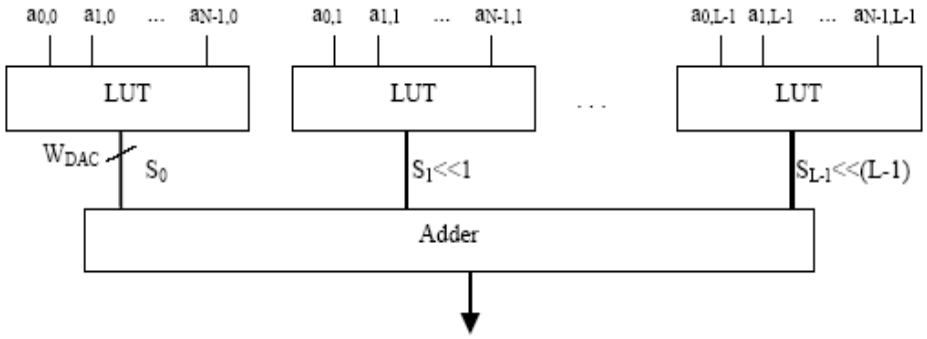


Figure 3.7 Diagram of Distributed Arithmetic Convolver

In comparison with the LC, the LUT data bus width of the DAC is smaller, as it can be seen from eq. 5-8.

$$W_{DAC}=K+\lceil log_2(N+1) \rceil \qquad W_{LC}= K+W_{IN}$$

where: WDAC - data width of LUTs for the DAC, WLC - data width of LUTs for the LC, WIN width of the input of the LUTs, K- width of the coefficients of the convolution, N- the size of the convolution kernel.

The data width of the LUTs is a direct sum for the LC, and is a sum of the logarithm of the number of inputs to the LUT for the DAC. This is a consequence that input bits are at the same significance for the DAC. The lower output width of the LUTs causes substantial FPGAs area savings, because not only smaller memory modules but also shorter adders are required. As a result, the DAC is preferable to the LC. The drawback of the DAC solution is that the dynamic change of the coefficient is much more difficult in comparison to the LC, which makes this approach rather impractical for dynamic systems.

A diagram of the DAC is shown in Figure 5-4. Similarly as for the LM, the size of the LUT memory grows rapidly with the size of the convolution kernel *N*. Therefore the LUT memory should be split into two or more independent LUTs, and then adders employed similarly like for the LM. The split of the memory should be implemented with respect to the cost relation between different memory modules and adders.

Consequently, in some cases the LUT based Hybrid Convoler (LHC) [Wia00c, Wia00d] - the hybrid of the LM and DAC, may be implemented, as the optimum memory split issue is concerned. For example, for the 3 3 convolution *N=9=3 3*, coefficient width *K=8* and input width *L=8*, two different memory modules should be used: four and five input memory blocks (4+5=9), but the 32 1 memory module occupies twice the area of the 16 1 module. Therefore the alternative LHC may employ the DAC for *N=8* and a single LM. The cost for the pure DAC is 226 XC4000 CLBs and 209 CLBs for the LHC [Wia00c]. Therefore 17 CLBs are saved by the use of the LHC.

3.7.2 .Irregular Distributed Arithmetic Convoler (IDAC)

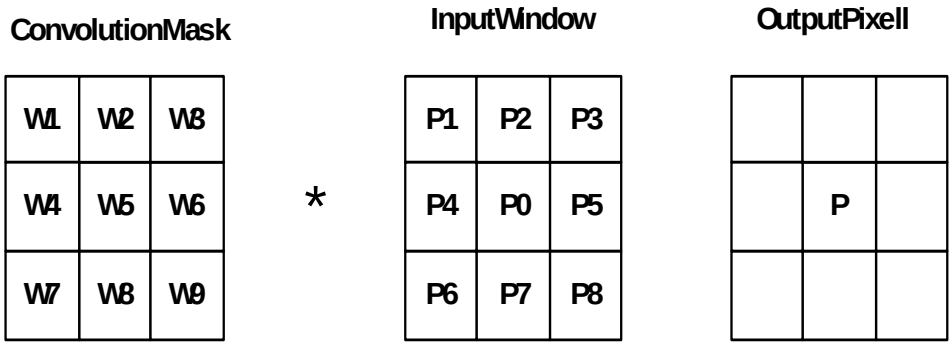
The previous solution assumes that the structure of the DAC is the same for different significance of input bits. However, this need not be the case, and bits of different significance can be grouped together in the same LUT. Therefore more or less a combination of the LC and DAC is obtained. This novel, introduced by the author of this thesis, design approach is denoted as Irregular Distributed Arithmetic Convoler (IDAC). An IDAC optimisation algorithm should optimise rather the address and data widths of memories and adder widths, and the bit-significance of inputs is only an input parameter which influences the LUT data widths.

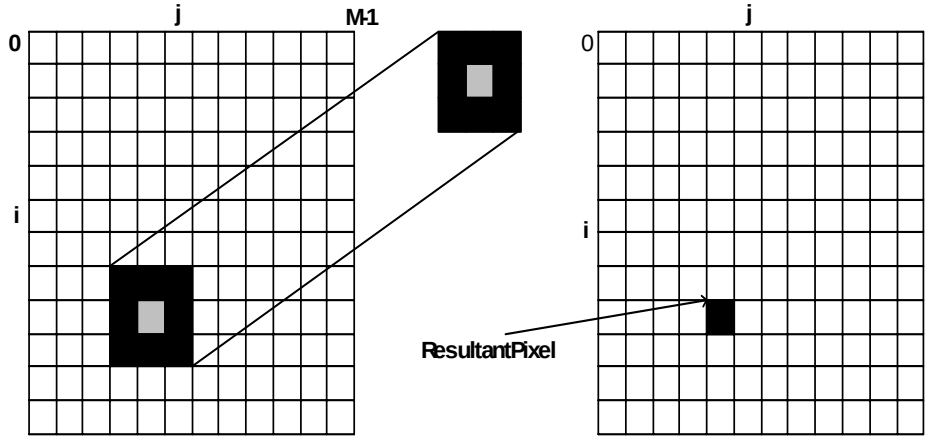
A greedy algorithm for IDAC is proposed. This algorithm optimises a partial solution, i.e. determines the LUT address width and the LUT inputs, according to the algorithm given in Listing 5-1. Before the optimisation algorithm is applied, every coefficient is shifted to the left until it is made odd. This reduces the data width of the LUT as the LSB of an even coefficient is fixed to zero. The input bit for which the coefficient is shifted is further treated as the input bit with significance increased by the number of shifts.

3.8 2-D Convolver

2D- Convolver operation

In image processing, a 2-D neighbourhood operation forms the basis of most low level image processing algorithms. Any neighbourhood image operation involves passing a 2-D window over an image, and carrying out a calculation at each window position, as illustrated in figure 3.8.





` **Figure 3.8.**The process of applying a neighborhood operation to an image

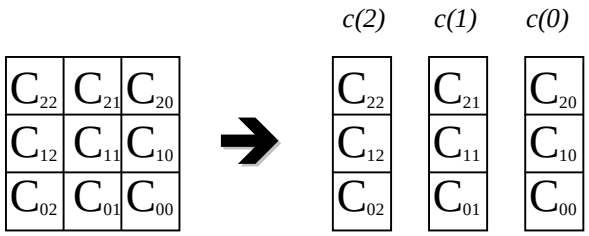
The result image is defined as follows:

$$R_{i+P-pt,j+Q-qt} = \sum_{l=0}^{Q-1} \sum_{k=0}^{P-1} C_{k,l} * X_{i+k,j+l} \qquad \forall i, j \in 0,0 \times M-P, N-Q$$

which can be rewritten as:

$$R_{i+pt,j+qt} = \sum_{l=0}^{Q-1} c(l)^T . x(i, j+l)$$

where *c(l)* are coefficient column vectors. For example, a 3 by 3 window may be reorganised as follows:



and *x(i,j+l)* are input image column vectors and where

$$c(l)^T . x(i, j+l) = \sum_{k=0}^{P-1} C_{k,l} * X_{i+k,j+l}$$

This has split the overall convolution into a sum of scalar-products. Each of these vectors of coefficients *c(l)* and column vectors *x(i,j+l)* produces a partial result. These can be calculated independently of the column *l* and finally added to produce the result value $R_{i,j}$.

To allow each pixel to be supplied only once to the FPGA, internal line buffers are required. This is a common approach used in many hardware realisation such as SH. The idea is that these internal line buffers (or line delays) are used to synchronies the supply of input values to the MAC units (multiply-and-accumulate) ensuring that all the pixel values involved in a particular neighbourhood operation are processed at the same instance. Figure 2.7 shows the architecture for a 2-D convolution operation with a 3 by 3 window kernel. It should be noted that several other architectures, such as Systolic Arrays, are available which also are capable of implementing such algorithms.

Contd

Each of the Processing Elements (PEs) stores a template weight (or coefficient) internally and performs the basic *multiply* and *accumulate* operations as defined in equation 2.3
above. The *line buffer* units are used to synchronise the 3 separate sets of PEs. The *word buffer (wb)* in each of the PEs is used to synchronise the supply of the pixels.

The Processing Element (PE) distinguishes a convolver from any of the other neighbourhood operations. Indeed, the initial structure of a PE has essentially only two varying components: a ***local operation*** (for convolution, a multiplier, a multiplier) and a ***global operation*** (for convolution, an accumulator). To implement the entire instruction set of the coprocessor, only a small set of (five) components blocks is required

Chapter 4

ALGORITHM IMPLEMENTATION

4.1 COMPLETE 3X3 CONVOLVER

The FPGAs dominates the DSP market with an over 50% market share . This processor was designed to handle the most computation intensive applications. However, even with single- cycle multiply-and-accumulate capability, certain low level operations, by their very nature, require several cycles to complete. The 2-D convolution, one of the most common image processing operations, belongs to this class of applications. In fact, a highly optimized VHDL language program, written to reform 3x3 convolution. even with the use of high performance features of the FPGA such as parallel instructions and delayed repeat blocks. With an accelerator targeting a throughput of one convolved pixel per cycle or more, this application offers a good potential for acceleration. The following section describes the implementation strategy chosen for a 3x3 convolution coprocessor.

4.2 3x3 convolution implementation strategy

The 3x3 convolution of an image is defined by equation 1:

$$P'_{m,n} = \sum_{i=-1}^{1} \sum_{j=-1}^{1} P_{m+i,n+j} \cdot W_{i,j}$$

where *P'm,n* is the convolved pixel, *Pm,n* is the image's actual pixel value, and *Wi,j* is the convolution kernel weight. Equation 1 indicates that the 3x3 convolution *P'm,n* of each pixel *Pm,n* requires knowledge of the values of its 8 immediate neighbors. Similar to the *Cytocomputer* machine , a strategy to extract windows of pixels from a single data stream has been adopted. Pixel values are fed line by line, from top to bottom, until 2 complete lines and the first 3 pixels of a third line are contained within a series of shift registers. At that point, all the pixels belonging to the first 3x3 convolution window are available inside the coprocessor.

Convolver Implementation Stratagy

From that moment on, each new pixel value inserted into the chain of shift registers effectively displaces the convolution window to a new adjacent position until the whole image has been visited (see **figure 4.2**).

Evidently, storing 2 complete 320 pixel lines within a chain of shift registers would be very expensive in an ASIC or FPGA based implementation. .. The problem with this scheme is that to compute 3x3 convolutions on band borders requires having access to pixel values belonging to adjacent bands. Hence, a certain amount of overlap must be allowed between bands. A number of pixel columns must then be transmitted more than once, thereby degrading the coprocessor's overall performance.

Furthermore, the 3x3 convolution kernel weights have been restricted to the values -4, -2, -1, 0, 1, 2 and 4. This set of values was chosen because it allowed several useful image enhancement filters to be implemented (average, Sobel, Prewiitt, Laplace, etc.), especially in the area of edge detection , while reducing by half the multipliers' overall size and complexity

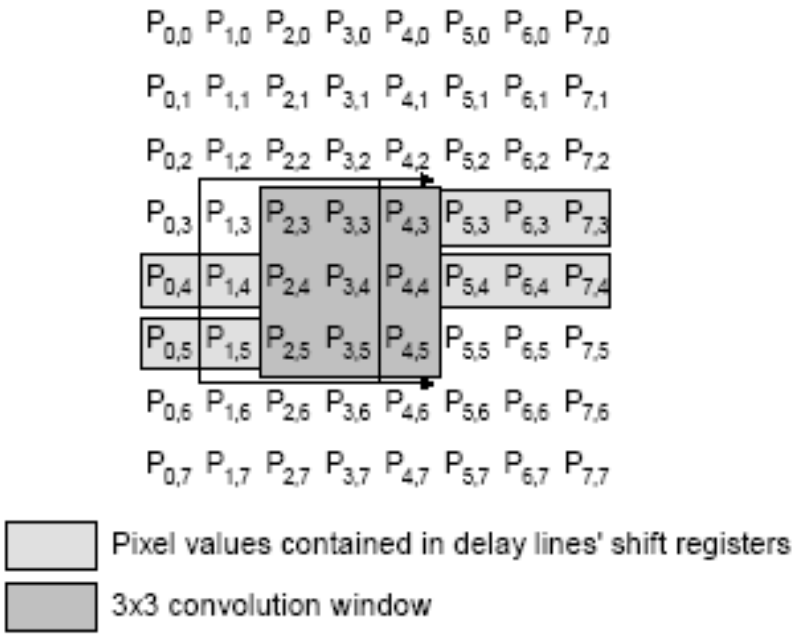


Figure.4.2 Displacement of Convolution Window

The reception of the pixel values is made through an FIFO of depth equals row size of the image that is 320 The input port's communication interface generates the necessary handshaking signals

required to complete data transfers. The control unit, composed of a finite state machine coupled with a counter and a comparator, keeps track of events and identifies each byte before it is read from the input FIFO according to a predetermined sequence.

4.3 3 x 3 Convolution Architecture

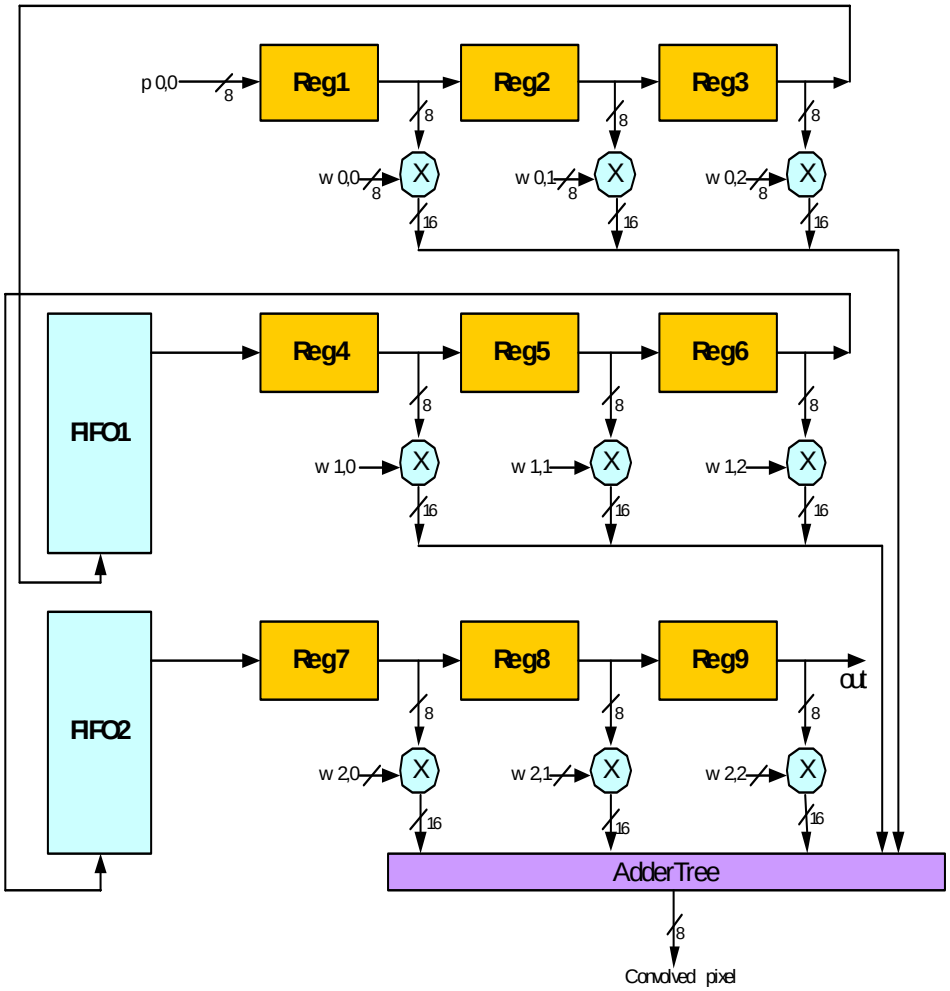


Figure 4.3 Architecture for 3x3 Convolver.

To compute a full 3x3 convolution in a single cycle, 9 multipliers were needed. With the supported kernel weights having been restricted to the aforementioned set of values, each multiplier's task could be reduced to single cycle shifts, resets to zero and two's complement conversions. A summation unit composed of carry-save adders in a Wallace-tree configuration tallies up the 9 pixel-weight products for each convolution window. The result produced by the

summation unit is then fed to a saturation module, which converts it back to an unsigned 8-bit value. A set of pipeline registers was inserted between the multipliers and the summation unit to preserve a 40 ns clock cycle.

Every element in this design was obtained through synthesis of VHDL behavioral descriptions. The Mentor Graphics 8.2.5 design environment provided all the compilation, simulation and synthesis tools required for the tasks . The synthesized netlist of the

Adder Tree Structure

convolution processor was mapped subsequently to Xilinx FPGA devices. Transistors and configurable logic block (CLB) counts for both mappings are provided .

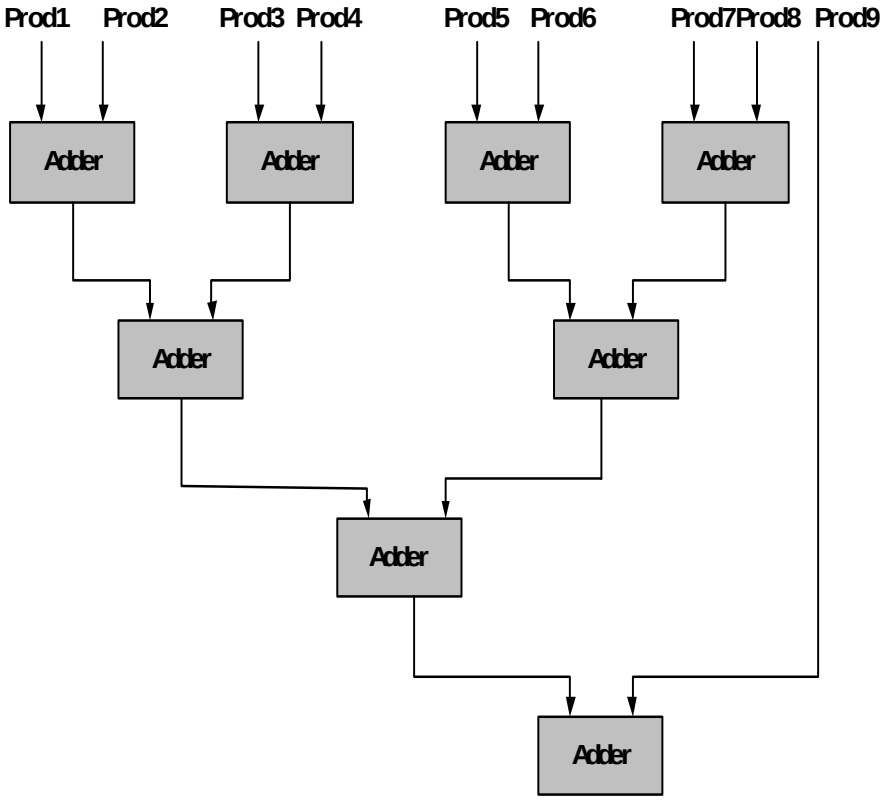


Figure 4.3.1 Adder Tree Structure

We included the and Xilinx Spartan II CLBs. The entire convolution coprocessor uses up 955 Slices and fits within Xilinx XC2S200 device,. A novel property of Xilinx Spartan II devices is the ability to use the look-up tables inside their CLBs as RAM. shift registers with RAM instead of

flip-flops. This implementation specific optimization, however, would not have eliminated the need for XC2S200 devices, and was not included in the present version of the convolution coprocessor for lack of time. Taking into account the communication ports' throughput and the effect of vertical band processing, a speed-up of over 14 can be expected compared to a software implementation.

With the experience gained from the design of the 3x3 convolver, we will see how an arbitrary-size 2-D convolver can be obtained. The object of this study is to find the most

convenient way of including a generalized 2-D convolver in a library of reconfigurable hardware accelerators.

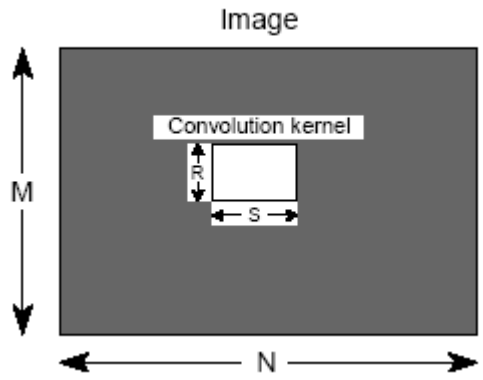


Figure 4.3.2 An MxN image processed using an RXS convolutional kernal

4.4 Synthesis Results:

Number of Slices: 955 out of 2352 40%

Number of Slice Flip Flops: 1689 out of 4704 35%

Number of 4 input LUTs: 1306 out of 4704 27%

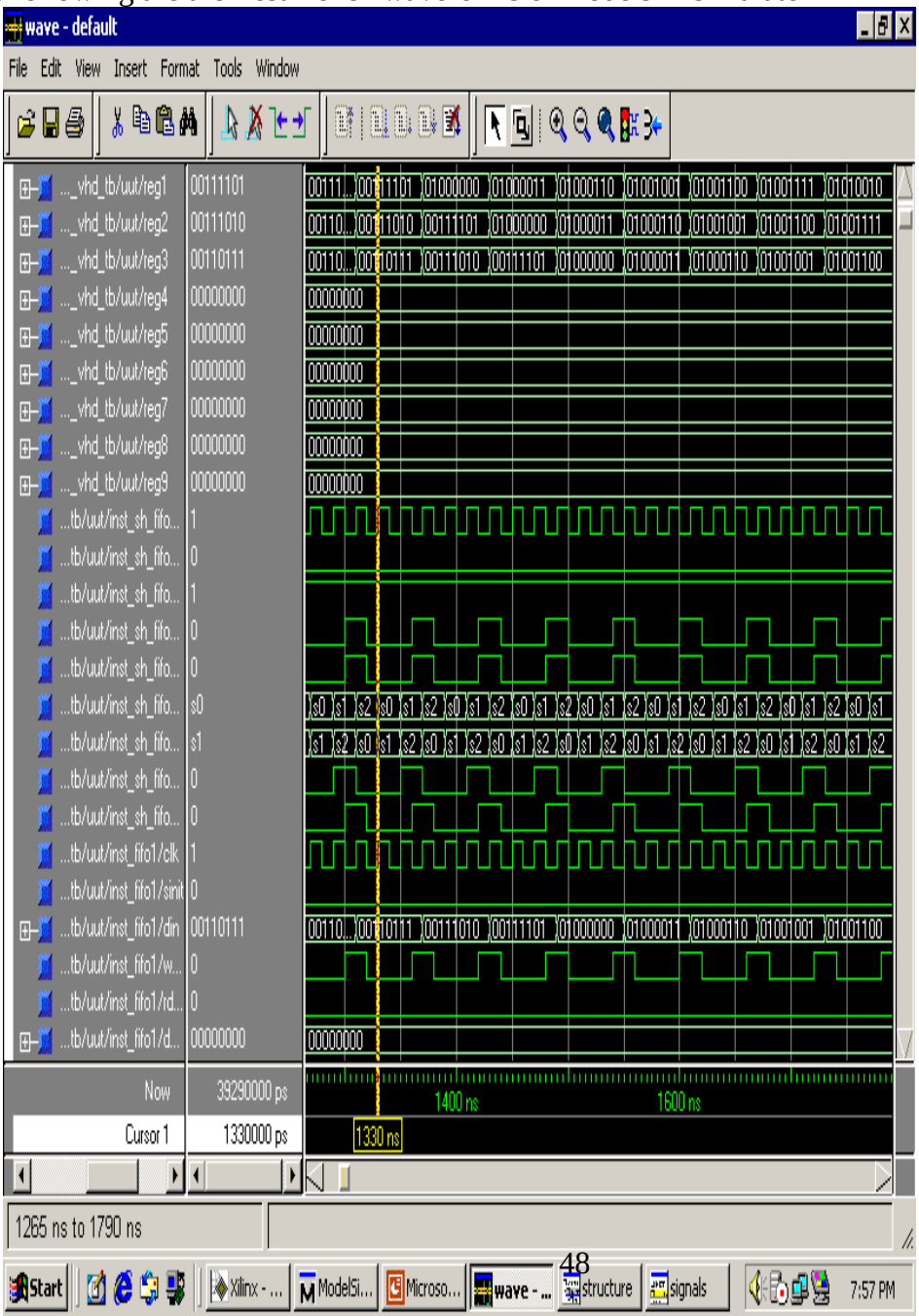
Number of bonded IOBs: 109 out of 144 75%

Number of BRAMs: 2 out of 14 14%

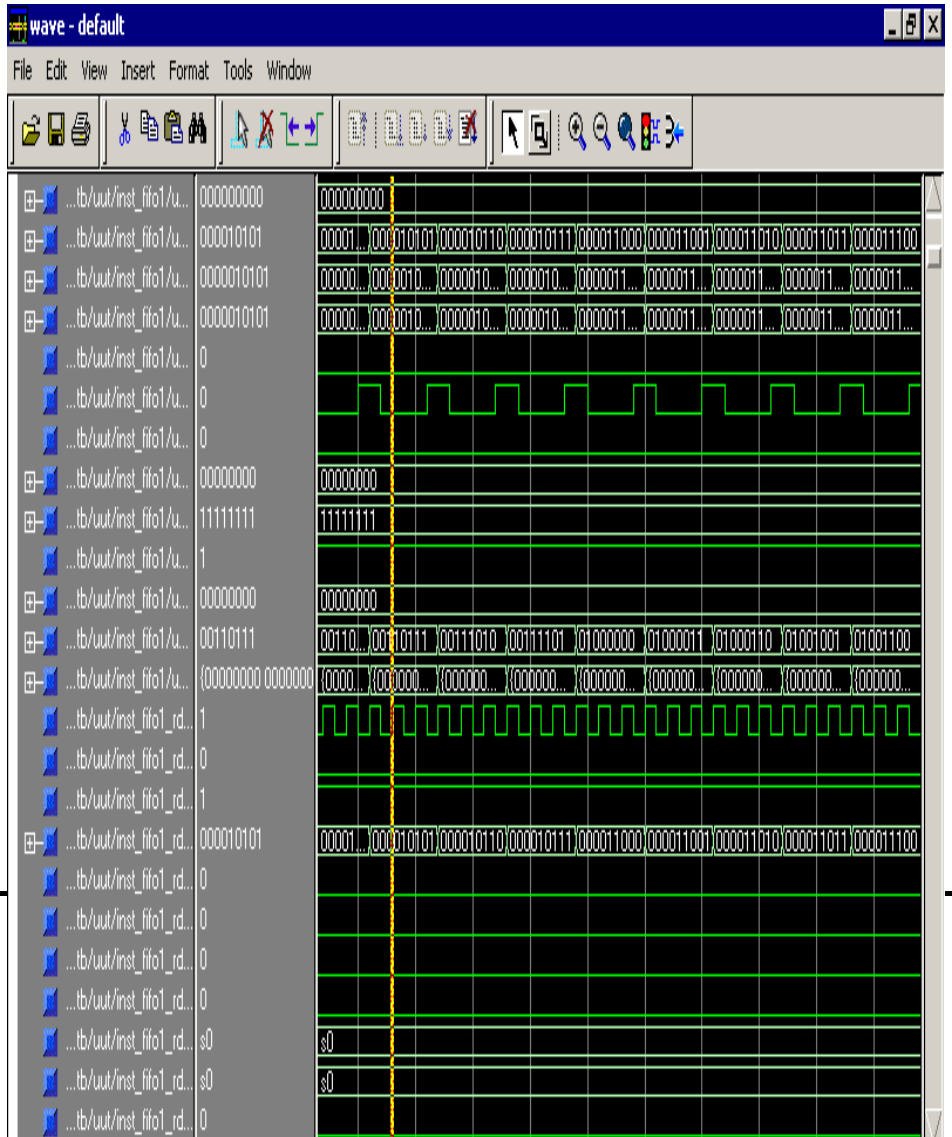
Number of GCLKs : 1 out of 4 25%

Test bench waveforms

The following are the Test Bench waveforms of modelsim simulator



Test bench waveforms



PCI Based Data Acquisition Card

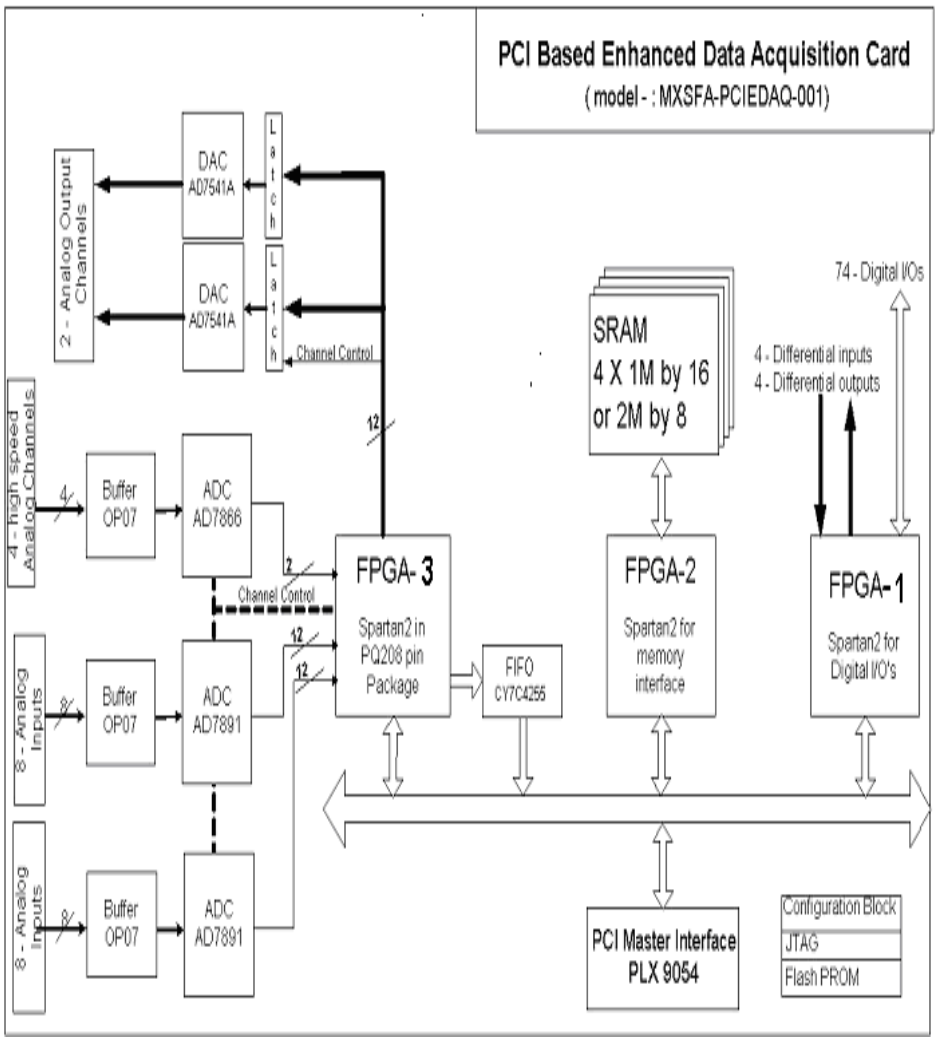


Figure 4.4 Enhanced Data Acquisition Card

After designing and verifying the 2D-Convolver now its needs made available for real time applications. The above figure shows arrangement for how it is implemented The above figure is the PCI based Data Acquisition Card, in this there are three Spartan II FPGAs which are used three different functionality. It is basically used for image processing applications as Capturing of image and FPGAs are used further processing such as image enhancement.

After designing of Convolver take the image of 320 x 240 stored in any one of the SRAMs provided on the board. From the SRAM read the image to FIFO inside the FPGA and the

Image Waveforms

resultant image pixel is again stored back into the SRAM. From the SRAM we read the image through the PCI to PC.

The following are input and output images

Input Image

Figure 6 Input Image

Vertical Edge Detection

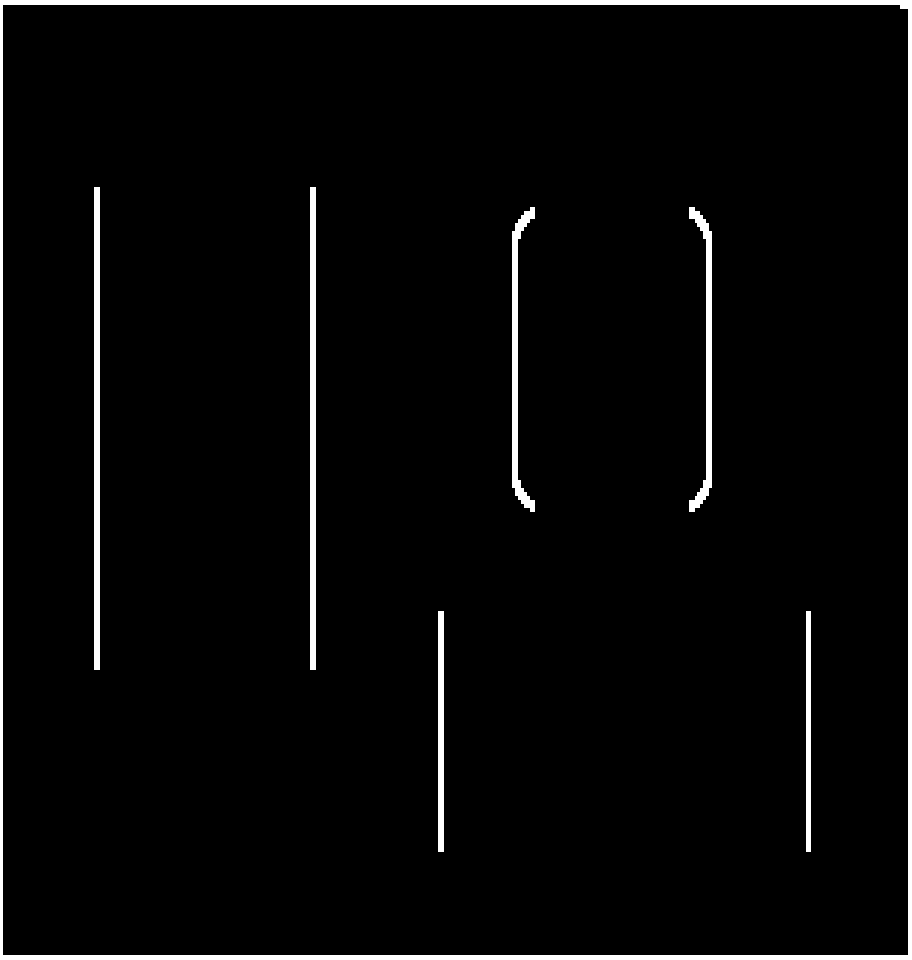


Figure 7 Vertival Edge Detected Image

Horizontal Edge Detection

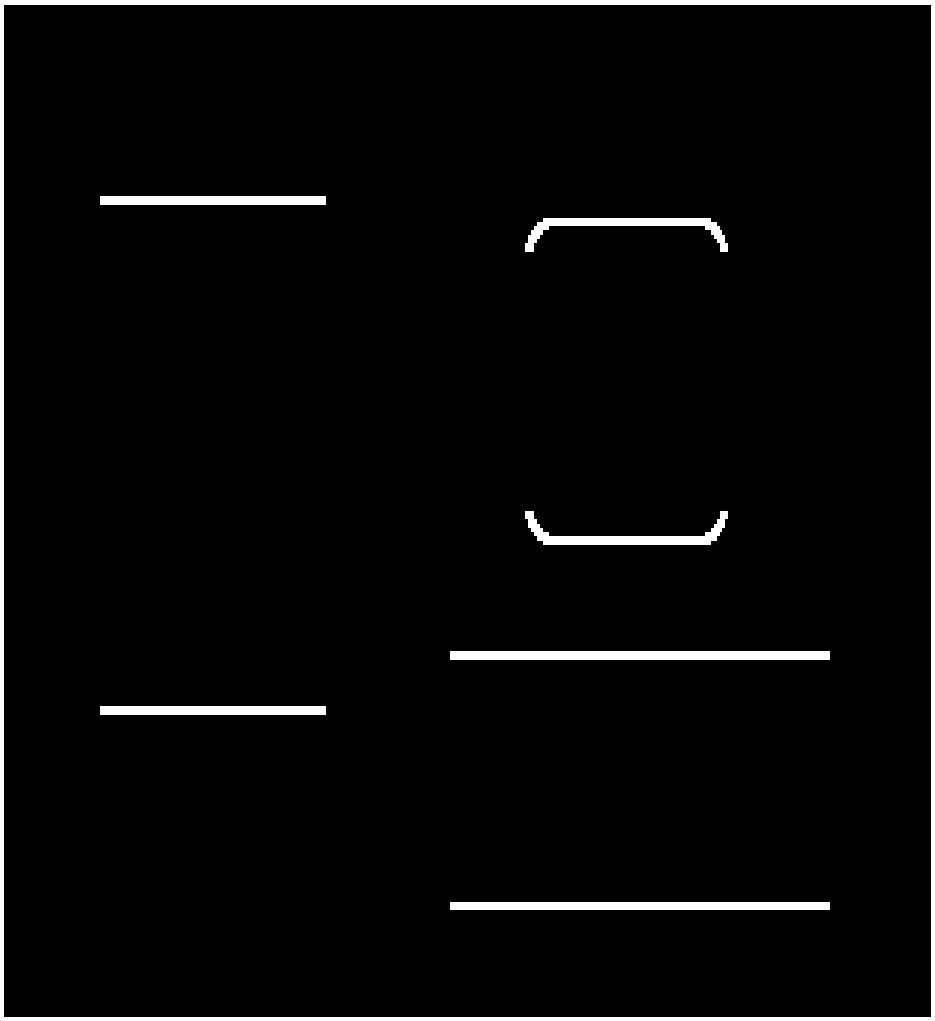


Figure 8 Horizontal Edge Detected Image

Input Image

Figure 9 Lina Input Image

Figure 10 Edge Detected Image

Input Image

Figure 11 Input Image

Figure 12 Enhanced Image

Chapter 5

Conclusions And Future Work

Conclusions

In this paper, the design of a new high-speed energy efficient FPGA-based 2-D 3x3 variable-precision convolver has been presented. The new architecture is extremely flexible. In fact, it can run-time adapt itself to perform two dimensional convolutions between variable bit-resolution image pixels and kernel weights avoiding time and power consuming reconfiguration processes. This is a very nice feature, which makes the new circuit particularly useful to support modern image processing algorithms that require adaptive filtering also for variable bit-resolution image pixels.

Future Work

The performance of the image processing algorithms in this work is achieved by implementing the algorithm on Field-Programmable Gate Arrays (FPGAs). Hardware implementation accelerates the designs by performing the operations concurrently. On the other hand, reprogrammability of the FPGAs allows for faster and cheaper design cycle of the system compared to Application Specific Integrated Circuit (ASIC) design. This work can be further extended to for real time implementation of object detection. The implementation of the whole design on FPGA is tedious and time consuming. With The advances in the software tools and growing functionality and capabilities of the FPGAs, hardware software co-design can be done, that drastically reduces the design time for high speed applications.

One of the current shortcomings of the designs presented in this thesis is the resource utilization of FPGAs. This is mainly due to the FIFO units being used in the design. FPGA resource utilization can be greatly reduced by creating FIFO buffers on external RAM. A large part of the improvement possible in this design lies in the algorithms themselves. If the kernel for the convolution design were to be changed, the convolution algorithms would have increased functionality for changing the convolution kernels on the fly.

Chapter 6

References

1. John C. Ross. “Image Processing Hand book”, CRC Press. 1994.

2. Stephen D.Brown, R.J. Francis, J.Rose Z.G.Vranesic. Filed Programmable Gate Arrays, 1992.

3. Moore,M A DSP-based real time image processing system, Proceedings of the 6 th International conference on signal processing applications and technology, Boston MA”, August 1995.

4. Rolf F. Molz, Paulo M. Engel, Fernando G. Moraes, Lionel Torres, Michel Robert.

5. Design of a Classification System for Rectangular Shapes Using a Co-Design

6. Environment, 13th Symposium on Integrated Circuits and Systems Design, 01/01/2000, pp. 281-286.

7. Xilinx Spartan II Field Programmable Gate Arrays. Production Product specification (V2.4) July 17, 2002.

8. Digital Video & Image processing Xilinx solutions for the Broadcast Chain. XilinxLtd 2002.

9. Richard G.Shoup. Parameterized Convolution Filtering in a Field programmable Gate Array Interval, Research Palo Alto, California .1993.

10. 3x3 Convolver with Run-Time Reconfigurable Vector Multiplier in Atmel AT6000

11. FPGAs. AT6000 FPGAs Application Note 1997.

12. V.Gemignani, M. Demi, M Paterni , M Giannoni and A Benassi. DSP

13. implementation of real time edge detectors. Proceedings of speech and image

processing pp 1721-1725,2001.

Chapter 7
System Review

FFTs are very common, computationally intensive signal processing functions found in a large number of signal processing systems. Orthogonal frequency division multiplexing (OFDM) systems, for example, require large amounts of FFT computing ability to handle the required data rate of all transmit and receive channels passing through the network. Typically, designers building OFDM transmitters and receivers have relied on digital signal processors as their device of choice for implementing these signal processing functions. DSPs typically come with a range of basic,assembly-optimized signal algorithms like FFTs and finite impulse response (FIR) filters, making these functions easier to implement compared to an ASIC or FPGA hardware-based approach. Unfortunately, the evolution in performance of DSPs has not been able to keep up with the demands of current and future communication system requirements, forcing designers to implement arrays of digital signal processors simply to satisfy the data rates and vast number of channels needed by the system. A common challenge arising from this approach involves handling shared memory between the processors and ensuring that data does not get overwritten from one processor to another. These arrays also tend to take up a lot of board real estate, and the large number of digital signal processors required can dramatically increase overall system cost.

An alternative approach to address this computational burden is to implement co-processors to speed up the computation of these functions using hardware accelerators. DSP vendors like Texas Instruments have started to add dedicated hardware co-rocessors to their DSP device offerings as seen in the C6416 DSP, which has dedicated on-chip Turbo and Viterbi co-processor hardware, primarily targeted at 3G wireless applications [1]. While this approach is beneficial for 3G applications, other users may not find these co-processors particularly suitable for their needs. DSP vendors, on the other hand, are driven to implement co- processors suitable for a broad-based market that is relatively mature.

Altera’s approach is to provide designers the flexibility to implement their co-processor on an FPGA [2]. These coprocessors can be suitably designed to fit virtually any function or application the user is targeting owing to the flexible nature of the FPGA’s device fabric. Additionally, designers are able to customize and construct their function in a way that fully

exploits the parallel nature of a hardware implementation within the FPGA, enabling better canalization (useful in communication systems), and ultimately, greater data throughput.

A large variety of FFT algorithms are available to the scientists that use the conventional serial or vector computers. However for parallel machines, the choice of FFT algorithms is very limited. In applications such as the pseudospectral methods for solving partial differential equations (PDE's), a number of multidimensional FFT's are computed per time step. The speed of the FFT computation is therefore very critical to any large application using the pseudospectral method. Since such very large computations are feasible mostly on only parallel machines, there is a need for fast multidimensional FFT algorithms for parallel machines.

The approach to computing multidimensional FFT's on parallel machines is currently under debate. There are two methods that are possible. One of the approaches is the ``Transpose Method". In this method, data are divided by planes between nodes. For example, in the three dimensional transform, each node has a number of planes on which it computes two dimensional FFT's. Next, a distributed transpose rearranges the data in such a way that the FFT along the third dimension can be computed locally. The parallel aspect of this approach is limited to the distributed transpose, which is equilvalent to a standard exchange problem [1]. Here, each node sends data to and receives data from all other nodes during distributed transpose. This method is fairly easy and has been implemented for a number of applications .

The second approach is to design a distributed FFT algorithm which operates without collecting planes or rows on a single node. Here the internode communications are interspersed with the computation at different stages of the FFT. Such an algorithm is more difficult to design and implement since the parallelization is an integral part of the algorithm. It gives a clear advantage of flexibilty in data distribution, since parallelization is possible along more than one dimension. In this article we present an algorithm which takes the second approach. The algorithm is based on the single dimensional distributed FFT and is designed to run efficiently for block scattered data

distributions. A three dimensional block scattered data distribution is defined by a data size $n_x \times n_y \times n_z$, a block size $b_x \times b_y \times b_z$ distributed over $p_x \times p_y \times p_z$ processors. The data are divided into equal sized blocks which are distributed over the processors in a wrap around fashion. An example of a block scattered data distribution is shown in Figure 1. The algorithm described here can compute FFT in one,

two or three dimensions for different blocksizes along different directions. The data could be parallel along one or more dimensions in any combination. A real-to complex FFT (RFFT) can be computed as a special case of this algorithm, where at least one dimension is non parallel. The non parallel dimension of the RFFT is computed within the node, and the computations for the remaining dimensions are similar to those in the CFFT algorithm.

Distributed Algorithm

The distributed multidimensional FFT algorithm described here computes forward and inverse Fourier Transforms. It is most suited for applications such as the pseudospectral method where the data are repeatedly transformed back and forth between configuration and phase space. This is because, the forward FFT redistributes the data across processors such that the final data distribution is completely different from the original one. Two factors contribute to the rearrangement of data in the forward transform. One factor is the reordering inherent in the basic FFT algorithm itself, and the other factor is reordering introduced by the parallelism. To get the original distribution back, extensive (possibly long range) data exchange is necessary, which would introduce significant communication overheads [2]. However, the inverse FFT is designed such that it restores the original distribution without introducing any overheads. Hence, a complete cycle of forward and inverse transform has identical data distribution at input and output. The reordering inherent in the FFT algorithm is countered by using different classes of FFT algorithms as the basis for each direction. The forward transform uses the Decimation in Frequency (DIF) algorithm and the inverse transform uses the Decimation in Time (DIT) algorithm. The two classes of the FFT algorithm are described here briefly.

DIF-FFTAlgorithm

For the DIF-FFT algorithm the N-point sequence **x(r)**is divided into two halves, $\mathbf{x_1(r)}$and $\mathbf{x_2(r)}$so that the the transformed sequence can be written as

$$X(2k) = \sum_{r=0}^{(N/2-1)} [x_1(r) + x_2(r)]\omega_N^{2kr} \quad (1$$

$$X(2k+1) = \sum_{r=0}^{(N/2-1)} [x_1(r) - x_2(r)]\omega_N^r \omega_N^{2kr}, \; k = 0, 1..N/2 - 1. \quad (2$$

These equations represent two **N/2** point DFT's of sequences $\mathbf{x_1(r)}$ + $\mathbf{x_2(r)}$ and. {$\mathbf{x_1(r)}$ - $\mathbf{x_2(r)}$] **W** $_{\mathbf{N}}$ **r**The process is then repeatedly applied to the two subsequences.

DIF-FFT **Algorithm**

For the DIT-FFT algorithm the N-point sequence **x (r)**is divided into two **N/2**-point sequences $\mathbf{x_1(r)}$ and $\mathbf{x_2(r)}$ as the odd and even elements of **x(r)** respectively; i.e.

$$x_1(r) = x(2r), r = 0, 1, 2, ..N/2 - 1, \quad (3)$$

$$x_2(r) = x(2r+1), r = 0, 1, 2, ...N/2 - \quad (4)$$

We then recursively compute $\mathbf{X_1(k)}$and $\mathbf{X_2(k)}$, the DFT's of $\mathbf{x_1(r)}$ and $\mathbf{x_2(r)}$respectively. The recursion stops when the DFT

of a 1-point sequence, which is the element itself, is required. The two sequences $\mathbf{X_1(k)}$and $\mathbf{X_2(k)}$ are then merged to generate **X (k)**using the following expressions

$$X(k) = X_1(k) + \omega_N^k X_2(k), \quad 0 \le k < N/2, \quad (5)$$

$$X(k) = X_1(k - N/2) - \omega_N^k X_2(k - N/2), \quad N/2 \le k < N. \quad (6)$$

The graphical representation of Equations and Equations is referred to as butteflies, and are shown in Figure . They are the basic unit of computation is an FFT algorithm. The sequence of data computations in an FFT algorithm can be shown graphically by a combination of butterflies, known as the flowgraph. The flowgraphs for the DIT-FFT and DIF-FFT for input sequence of length **8** are shown in Figure . Notice that the input and output data ordering of the two classes of FFT algorithm are complimentary to each other. DIF-FFT requires the input sequence to be in-

order and generates the output sequence in bit-reversed order while the DIT-FFT takes the input in bit-reversed order and generates the output in-order.

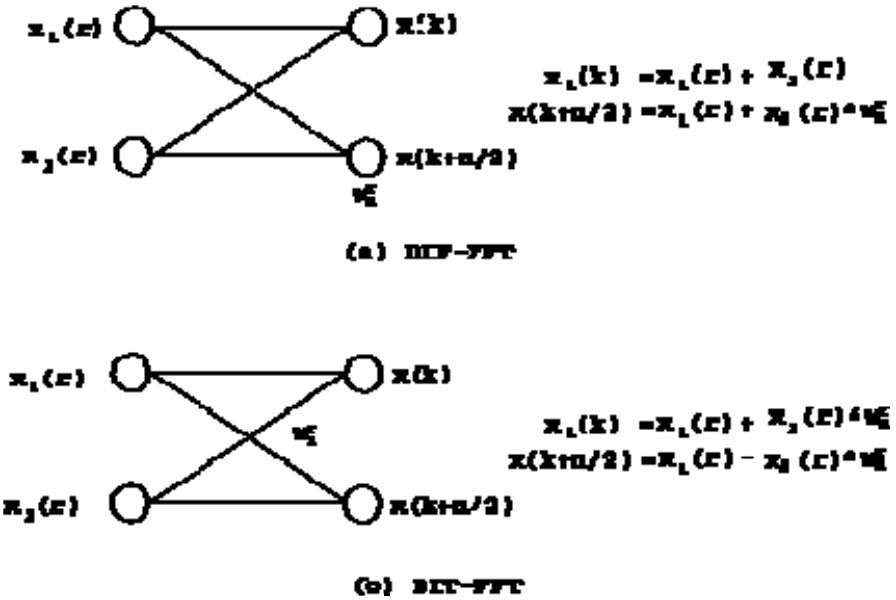


Figure 2: Butterflies for decimation in frequency and decimation in time FFT's.

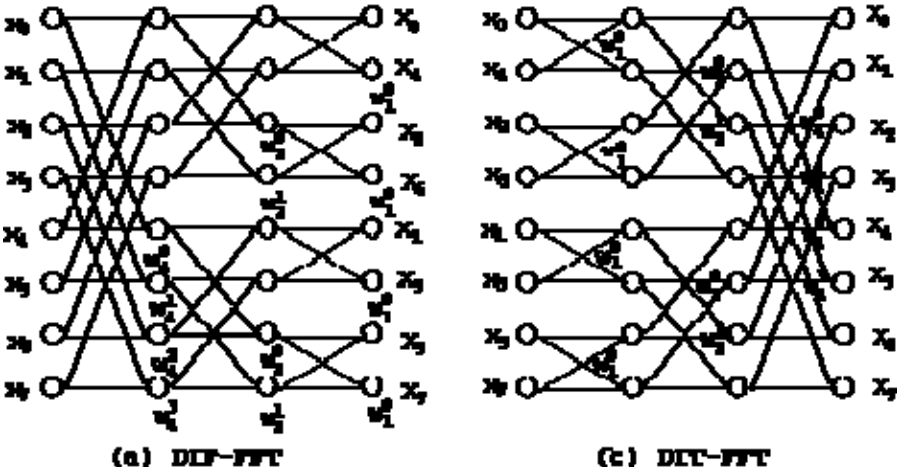


Figure 3: Flowgraphs for 8-point FFT's.

The data rearrangement due to parallelism is introduced by the butterfly computation. At any stage in the FFT, internode communication is required when the two data points forming a butterfly are at different nodes. In such a situation, to avoid rearrangement, the computation of butterflies would have to be done as shown in Figure . However, this approach has several disadvantages, pointed

out in fig. These are : high communication volume (all the data from one node sent to another one), unbalanced load (half the nodes have a multiplication and an addition per butterfly, while the other half have only one addition) and extra storage (the receiving buffer size is the same as the data size). By moving the data prior to computation as per Figure 4(b) these disadvantages are overcome at the cost of causing some data

rearrangement. However, since the data are already rearranged due to the inherent characteristic of the FFT algorithm, this is an insignificant cost. The section on data movement explains how the reordering introduced by parallelism is countered in the reverse transform.

Chapter 8

Algorithm Implementation

In this, for the implementation of 16-point FFT I have used the radix-4 as Basic Butterfly structure. The VHDL implementation for the basic butterfly structure is given below. In this I have used the parallel architecture. Since I have used the radix-4 as a basic butterfly unit so there are only two stages. The advantage of using the parallel architecture is when we process the operation on the second stage the first stage remains idle, so this drawback can overcome by using this parallel architecture.

Initially the 16-point data is stored in the RAM, and we require 4 clock cycles to read the four data items, and we have to apply all the inputs to basic butterfly structure simultaneously. For that demux and register arrangement is required.

As mentioned previously, our aim was to design/implement a lightweight and fast FFT processor targeting Virtex-II FPGA. The high-level schematic of the processor is We initially wanted to use ISE for synthesis. ISE is a high-level synthesis framework that into a VHDL description However, as can be seen from the schematic, our core will accept streaming input. Therefore, it must have pipelined READ, EXECUTE and OUTPUT stages.However, we could not find a way to express *streaming* in C. This is because of no support for concurrency. So, we revised our project goal downwards and instead decided to write the FFT code in VHDL. Since, the FFT code was to be written in VHDL, we also had to change our design flow from using ISE for synthesis to using Xilinx ISE instead.

The original DFT is:

$$X(k) = \sum_{n=0}^{N-1} x(i) W_N^{kn}, 0 \le k \le N-1, W_N = e^{-j2\pi/N}$$

The radix-4 DIF FFT would be given by:

$$X(4k) = \sum_{n=0}^{N/4-1}\left[x(n) + x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) + x\left(n+\frac{3N}{4}\right)\right] W_N^0 W_{N/4}^{kn}$$

$$X(4k+1) = \sum_{n=0}^{N/4-1}\left[x(n) - jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) + jx\left(n+\frac{3N}{4}\right)\right] W_N^n W_{N/4}^{kn}$$

$$X(4k+2) = \sum_{n=0}^{N/4-1}\left[x(n) - x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) - x\left(n+\frac{3N}{4}\right)\right] W_N^{2n} W_{N/4}^{kn}$$

$$X(4k+3) = \sum_{n=0}^{N/4-1}\left[x(n) + jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) - jx\left(n+\frac{3N}{4}\right)\right] W_N^{3n} W_{N/4}^{kn}$$

Architecture of the 16-point parallel FFT architecture using Radix-4 as a basic butterfly unit is given below.

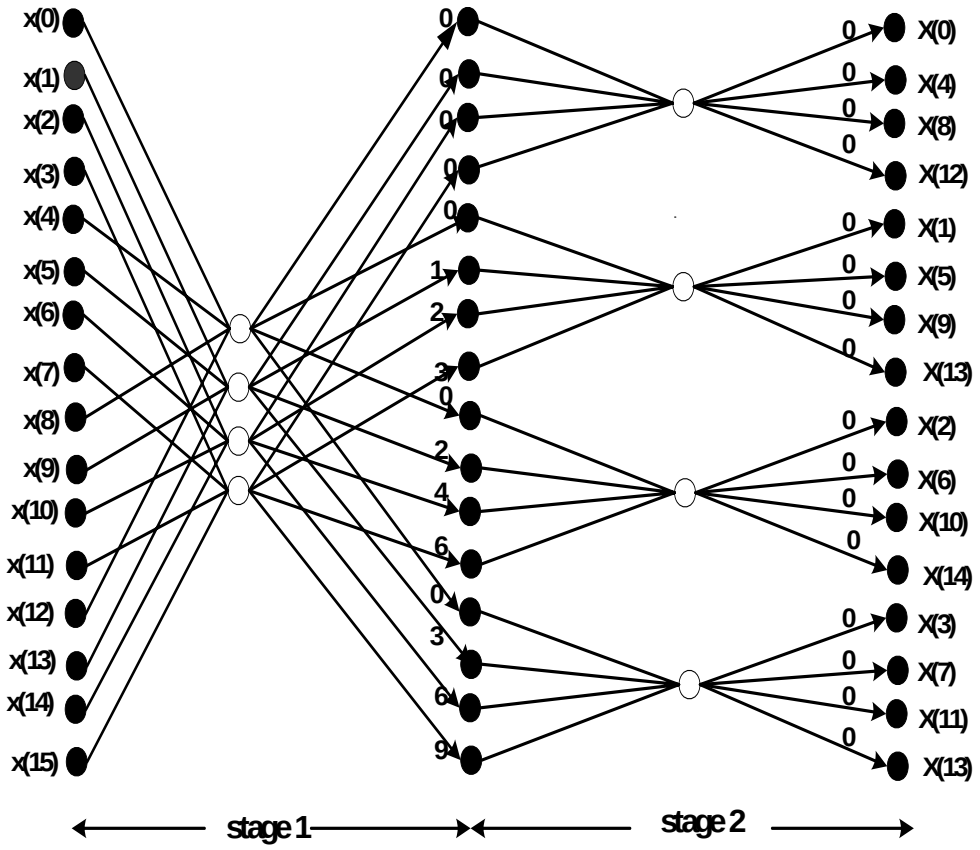


Figure8.1 flow diagram of 16-point FFT

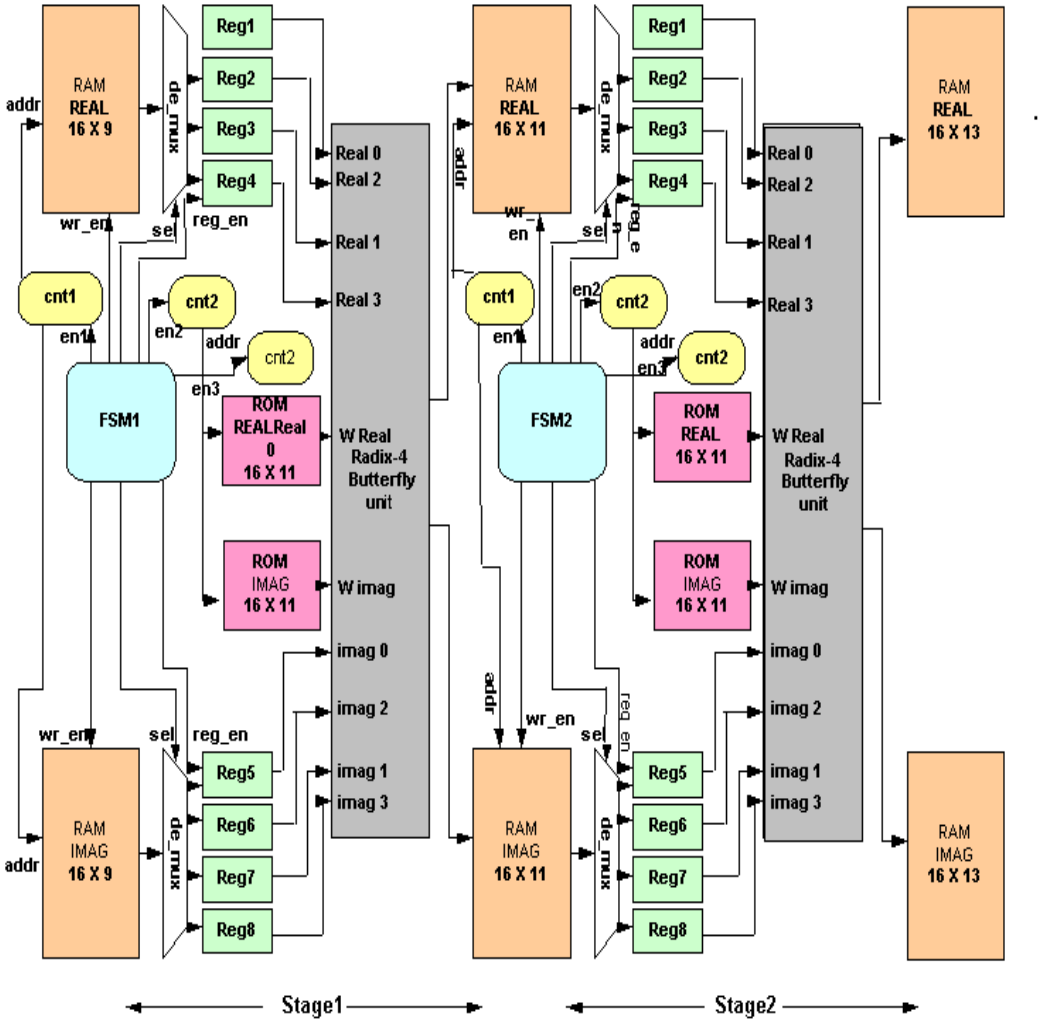


Figure8.2 parallel archirecture

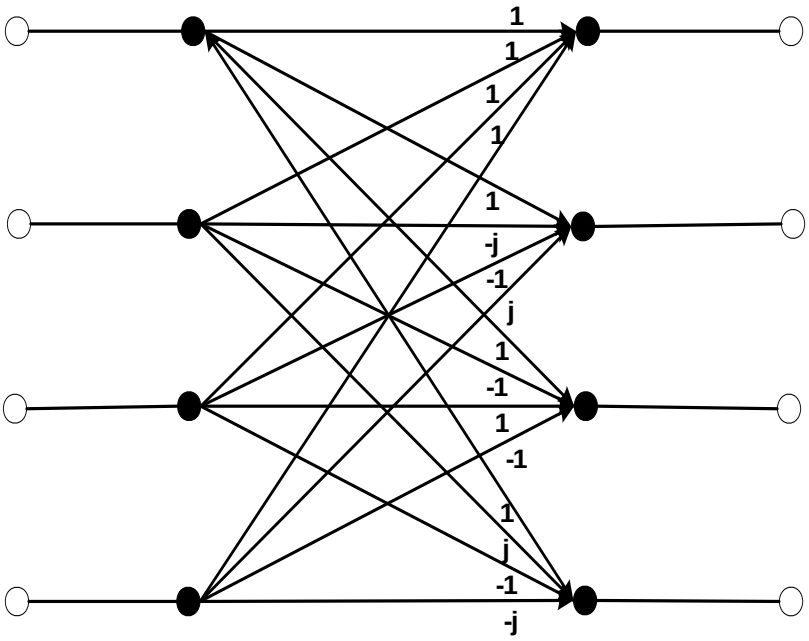
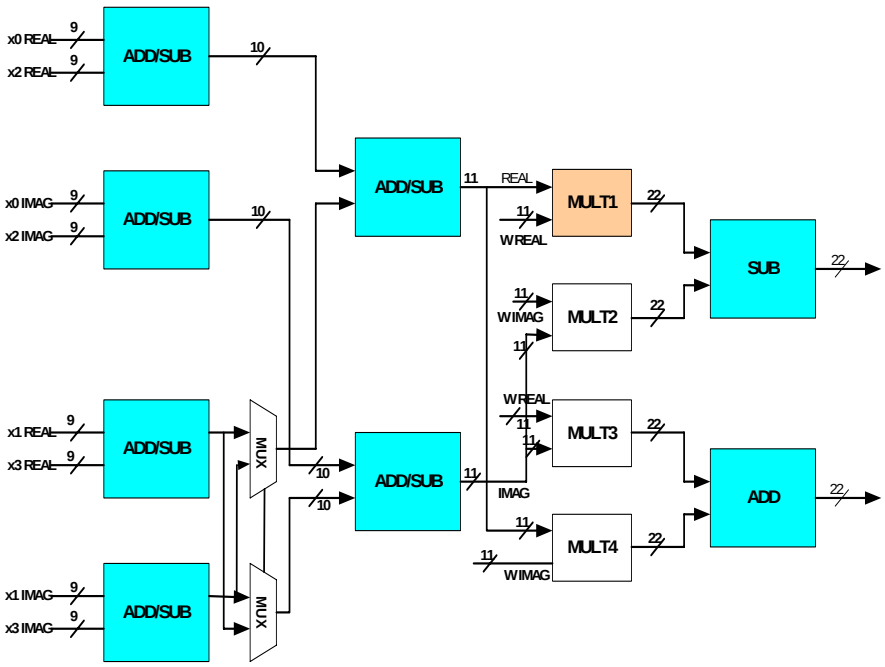


figure8.3 : basic radix-4 butterfly unit



Synthesis Results

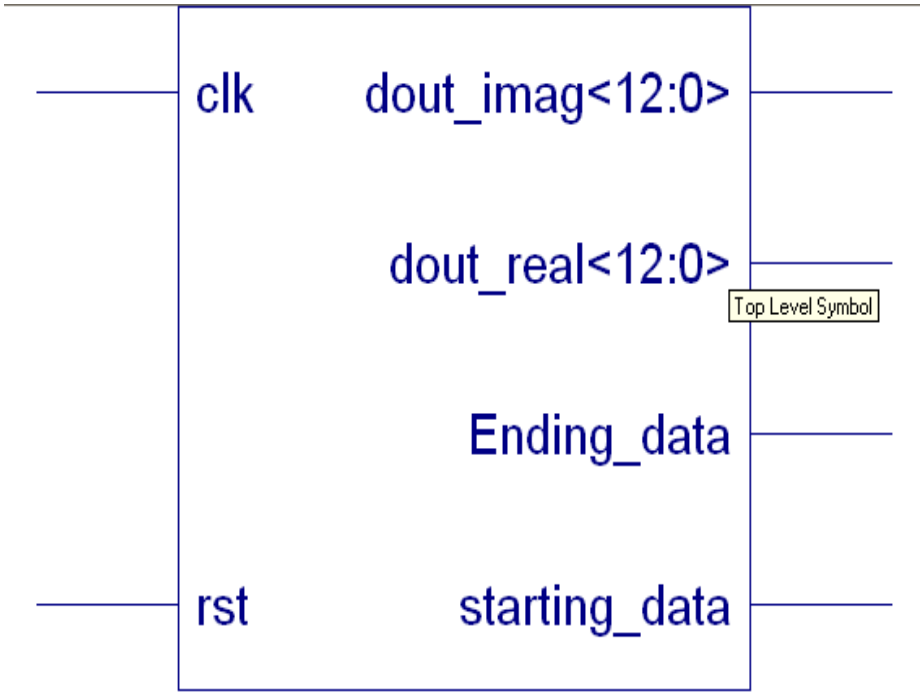
Number of Slices:	1261	out of 19392	6%
Number of Slice Flip Flops:	510	out of 38784	1%
Number of 4 input LUTs:	2106	out of 38784	5%
Number of bonded IOBs:	29	out of 692	4%
Number of TBUFs:	4	out of 10208	0%

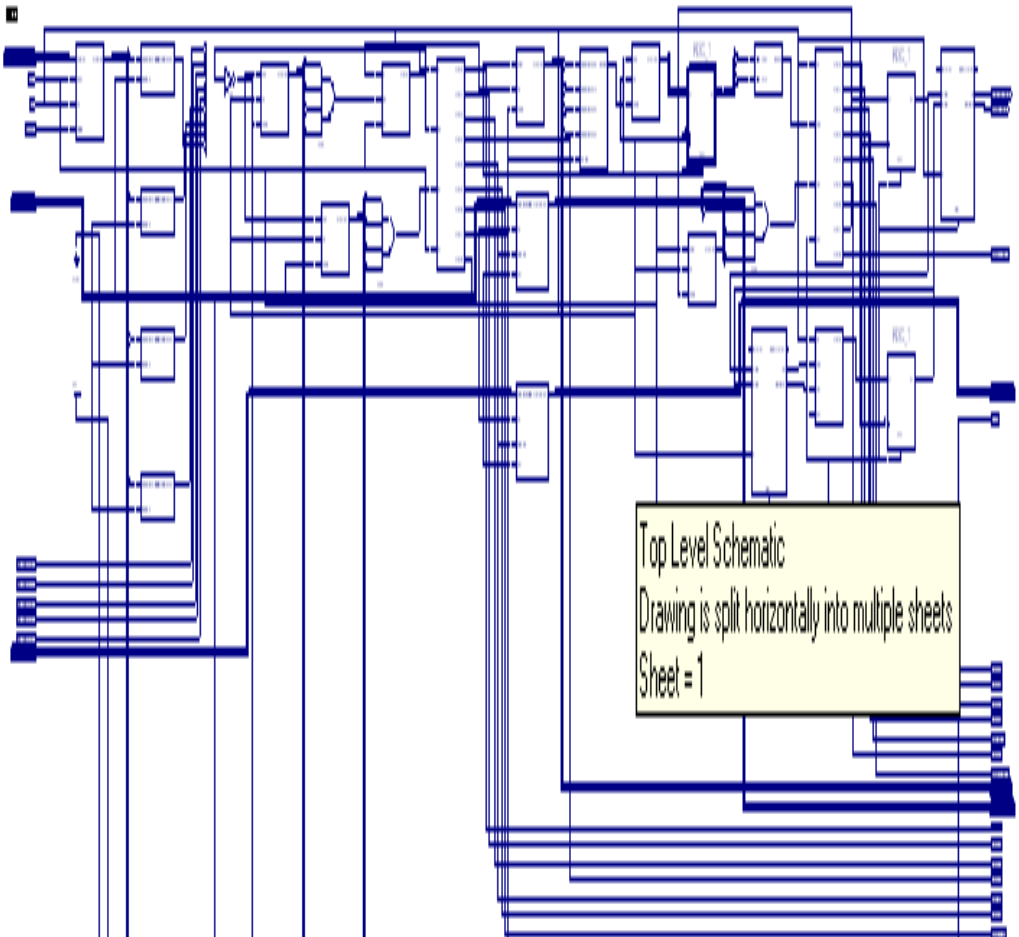
Number of BRAMs: 4 out of 192 2%

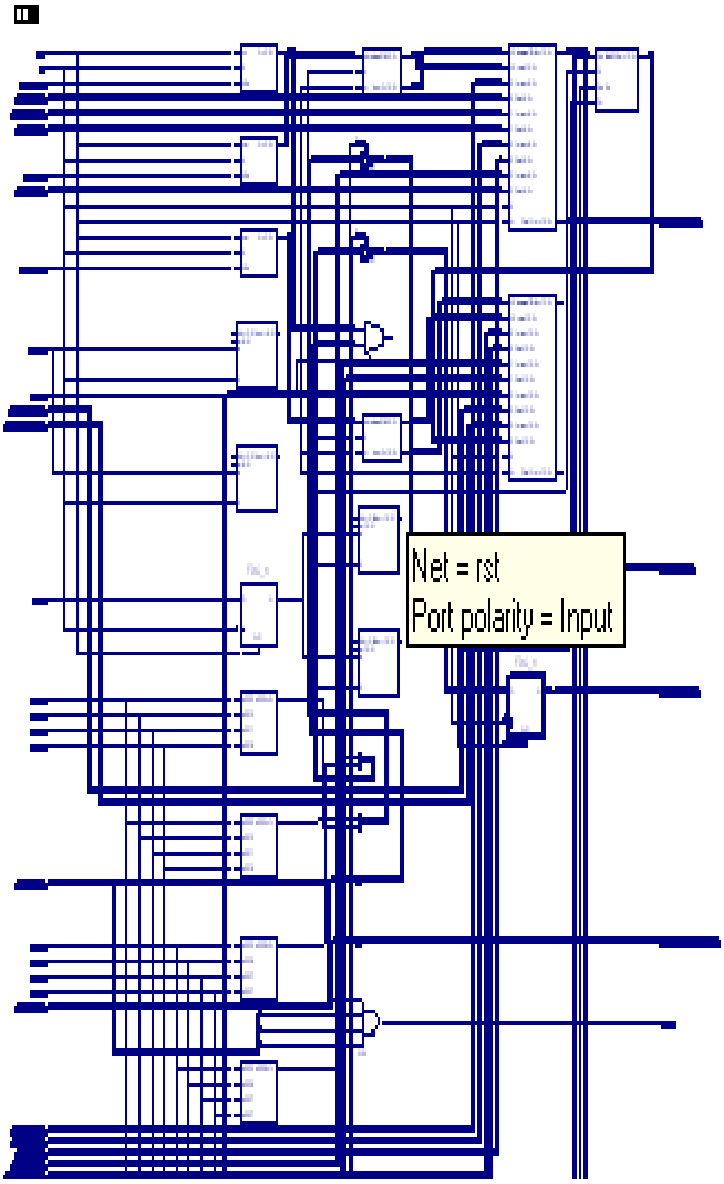
Algorithm implementation

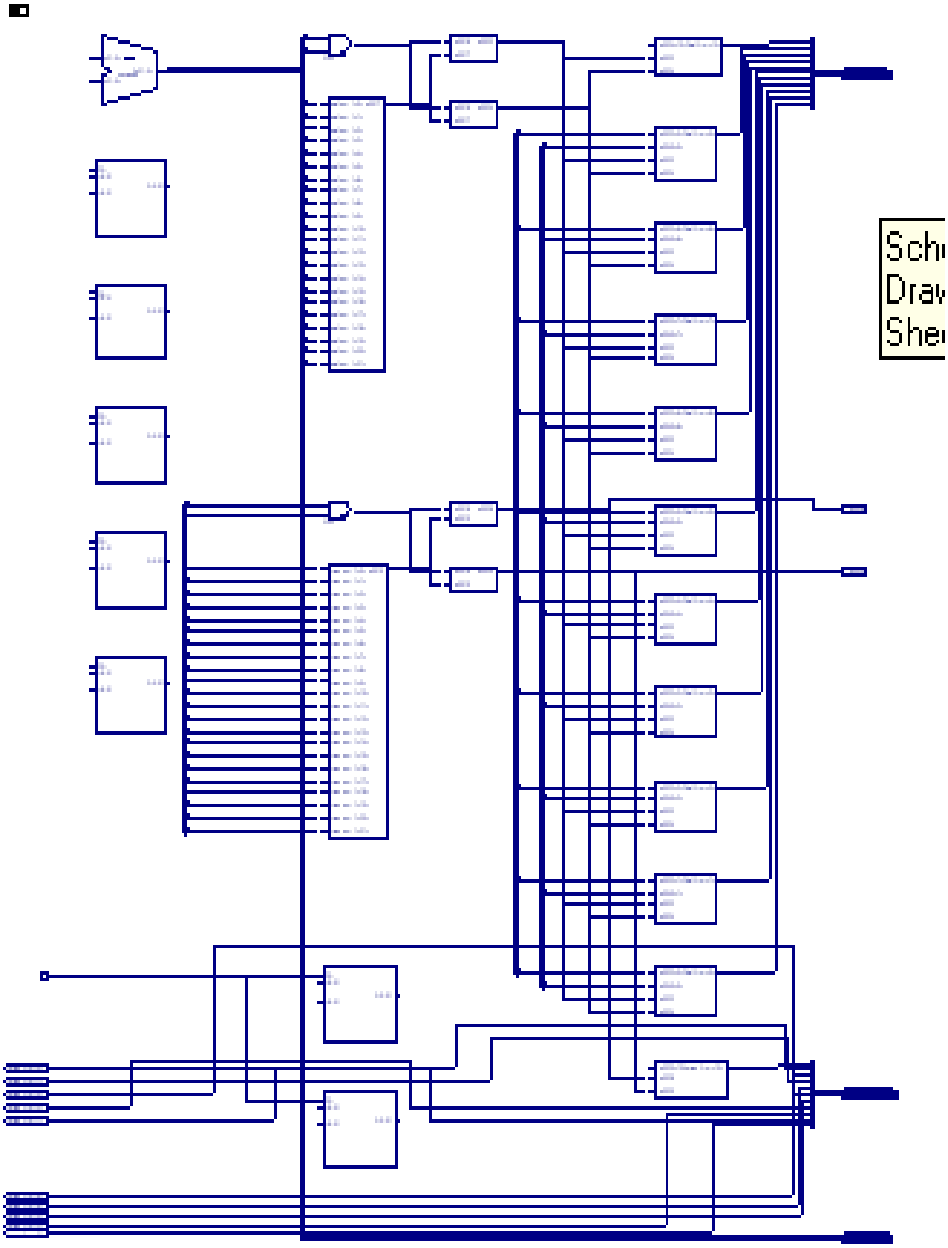
Number of GCLKs: 1 out of 16 6%

RTL schematic resulting from the Xilinx Synthesis Tool






Net = rst
Port polarity = Input



Chapter 9

SUMMARY & CONCLUSIONS

DSP and FPGA co-processor solutions provide designers with a myriad of implementation options and solutions for today's system designers. Along with these solutions comes a variety of design factors and considerations that need to be evaluated to select the best approach, depending on system requirements like ease of implementation, cost and performance as well as power consumption. Digital signal processors can provide the simplest implementation for a wide range of DSP algorithms and applications, but the cost/performance, implementation flexibility and hardware parallelism provided by an FPGA co-processor cannot be overlooked. A simple example of this is a basic OFDM communication system, which requires many FFT operations to be computed for a large number of channels at once.

Today's digital signal processors are unable to keep up with the load required of these systems unless an approach requiring large arrays of digital signal processors is employed. Integration of these digital signal processors within the system is not a trivial task, as the coordination of shared memory between processors is especially complex. FPGAs, on the other hand, are capable of parallelizing the operation of these functions, reducing the overall computation time of each operation and are, therefore, able to support a larger number of channels within a single device. Additionally, the densities of FPGAs have grown significantly over the last two years to the point that multiple instantiations of these functions can be implemented within a single FPGA, reducing the total number of devices required and ultimately board real estate. From a price/performance comparison, FPGA co-processors provide better performance for lower cost compared to a single digital signal processor approach. Additionally, because the FPGA is not fully utilized, more

functionality and arallelism could be added to the FPGA co-processor to increase the mount of processing the FPGA is capable of without impacting the cost of the system. Also, from a function-to-function power comparison, we see that for the same function, an FPGA implementation is capable of consuming less power than a digital signal processor. For comparable performance to an FPGA, a designer may be required to implement an array of multiple digital signal processors, something which could possibly increase the cost and power consumption of a system beyond that of an FPGA co-processor implementation.

References

[1] TMS320C6414, TMS320C6415, TMS320C6416 Fixed-Point Digital Signal Processing Data Sheet,Texas Instruments, February 2001.

[2] S. Lim, and P. Ekas, “Design Methodology for Hardware Acceleration for DSP”, Proc. International Signal Processing Conference, GSPx,2003.

[3] TMS320C64x DSP Library Programmer’s Reference, Texas Instruments, October 2003.

http://www.bdti.com/dspinsider/archives/dspinside r_040218.html

[4] Low Power Programmable DSP Chips: Features and System Design Strategies, BDTi Inc, 1994.

http://www.bdti.com/articles/info_icspat94lowpow er.htm

[5] Atlantic Interface Functional Specification, version 3.0, Altera Corporation, June 2002.

[6] FFT MegaCore Function User Guide, version 2.1.0, Altera Corporation, June 2004.

[7] Fast, Continuous, Sine Wave Generator, GlobalDSP, December 2003.

[8] Stratix FPGA Device Handbook, version 3.0, Altera Corporation, April 2004.

[9] TI Moves ‘C64x to 90 Nanometers, 1GHz, BDTi’s DSP Insider, Vol. IV, No. 2, February 18, 2004.

http://www.bdti.com/dspinsider/archives/dspinside

r_040218.html

[10] www.xilinx.com.

Appendix A

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity Convolution_2d is
   generic(width : integer:=8);
          Port (
                    clk                              :  in std_logic;
        rst                       :  in std_logic;
        COEF_REG1         :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG2         :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG3         :  in STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG4         :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG5         :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG6         :  in STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG7         :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG8         :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                    COEF_REG9         :  in STD_LOGIC_VECTOR(width DOWNTO 0);
                    start                    :  in std_logic;
        sram_wr                :  out std_logic;
                    data_in                :  in std_logic_vector(7 downto 0);
        data_out                :  out std_logic_vector(15 downto 0);
        valid                       :  out std_logic);

end Convolution_2d;

architecture Behavioral of Convolution_2d is
----------------Signals for first FSM1------
signal              rd_done : std_logic;
signal              wr_fifo1            : std_logic;
--signal            sh1_done            : std_logic;
signal              sh1_reg  : std_logic;
------------------------------Signals for fifo1-------------

signal              dout1 :  std_logic_VECTOR(7 downto 0);
signal              full1 :  std_logic;
signal              empty1:  std_logic;

-------------------------------- SIGNALS FOR fsm2
signal              fifo1_count         : std_logic_vector(8 downto 0);
signal              rd_fifo1            :  std_logic;
signal              sh2_done                    :  std_logic;
signal              sh2_reg             :  std_logic;
signal              wr_fifo2                    :  std_logic;
-----------------------------------------Fifo2---


signal              dout2 :  std_logic_VECTOR(7 downto 0);
signal              full2 :  std_logic;
signal              empty2:  std_logic;

-------------------------------SIGNALS FOR FSM3
```

```
signal		fifo2_count	: std_logic_vector(8 downto 0);
signal		rd_fifo2	:  std_logic;
signal		sh3_reg		:  std_logic;
signal		sh3_done		:  std_logic;

-----------------------------------SIGNALS FOR MAC

signal		Reg1 : std_logic_vector(7 downto 0);
signal		Reg2 : std_logic_vector(7 downto 0);
signal		Reg3 : std_logic_vector(7 downto 0);
signal		Reg4 : std_logic_vector(7 downto 0);
signal		Reg5 : std_logic_vector(7 downto 0);
signal		Reg6 : std_logic_vector(7 downto 0);
signal		Reg7 : std_logic_vector(7 downto 0);
signal		Reg8 : std_logic_vector(7 downto 0);
signal		Reg9 : std_logic_vector(7 downto 0);
--signal		Mac_out :  std_logic_vector(15 downto 0)
-------------------------------------First FSM ---

	COMPONENT sh_fifo1_write
	PORT(
		clk		: IN std_logic;
		rst		: IN std_logic;
		rd_done : IN std_logic;
		wr_fifo1	: OUT std_logic;
	--	sh1_done	: OUT std_logic;
		sh1_reg  : OUT std_logic
		);
	END COMPONENT;
-----------------------fifo1-----------------
component fifo IS
	port (
	clk	: IN std_logic;
	sinit: IN std_logic;
	din  : IN std_logic_VECTOR(7 downto 0);
	wr_en: IN std_logic;
	rd_en: IN std_logic;
	dout : OUT std_logic_VECTOR(7 downto 0);
	full : OUT std_logic;
	empty: OUT std_logic;
data_count: OUT std_logic_VECTOR(8 downto 0));
END component;
-------------------FSM2----------------------
	COMPONENT fifo1_rd_sh2
	Generic(N			:integer:=322);	---------No of columns in Frame
	PORT(
		clk			: IN std_logic;
		rst			: IN std_logic;
		rd_done		: in std_logic;
		fifo1_count	: IN std_logic_vector(8 downto 0);
		rd_fifo1		: OUT std_logic;
		sh2_done		: OUT std_logic;
		sh2_reg		: OUT std_logic;
		wr_fifo2		: OUT std_logic
		);
	END COMPONENT;

 ------------------------------FSM3----
	COMPONENT fifo2_rd_sh3
	Generic(N			:integer:=322);	---------No of columns in Frame
	PORT(
		clk			: IN std_logic;
```

```
                rd_done         : in std_logic;
                rst                     : IN std_logic;
                fifo2_count     : IN std_logic_vector(8 downto 0);
                rd_fifo2        : OUT std_logic;
                sh3_reg         : OUT std_logic;
                sh3_done                : OUT std_logic
                );
        END COMPONENT;
------------------------MAC---------
        COMPONENT reg_mac
        Generic(width:integer:=8);
        PORT(
                clk                             : IN std_logic;
                rst                             : IN std_logic;
                rd_done                 : IN std_logic;
                sram_wr                         : out std_logic;

                COEF_REG1       :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG2       :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG3       :  in STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG4       :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG5       :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG6       :  in STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG7       :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG8       :  IN STD_LOGIC_VECTOR(width DOWNTO 0);
                COEF_REG9       :  in STD_LOGIC_VECTOR(width DOWNTO 0);
                Reg1            :  IN std_logic_vector(7 downto 0);
                Reg2                    :  IN std_logic_vector(7 downto 0);
                Reg3                    :  IN std_logic_vector(7 downto 0);
                Reg4            :  IN std_logic_vector(7 downto 0);
                Reg5            :  IN std_logic_vector(7 downto 0);
                Reg6                    :  IN std_logic_vector(7 downto 0);
                Reg7            :  IN std_logic_vector(7 downto 0);
                Reg8            :  IN std_logic_vector(7 downto 0);
                Reg9            :  IN std_logic_vector(7 downto 0);
                Mac_out         :  OUT std_logic_vector(15 downto 0)
                );
        END COMPONENT;

begin

        rd_done<=start;
        valid<= start;
        Inst_sh_fifo1_write: sh_fifo1_write PORT MAP(
                clk             => clk,
                rst             => rst,
                rd_done  => rd_done,
                wr_fifo1        => wr_fifo1,
        --      sh1_done        => sh1_done,
                sh1_reg  => sh1_reg
        );

process(clk,rst,sh1_reg)
begin
if rst ='1' then
                Reg1<=(others=>'0');
                Reg2<=(others=>'0');
                Reg3<=(others=>'0');
elsif clk'event and clk ='1' then
        if sh1_reg ='1' then
                Reg1<= data_in;
                Reg2<= Reg1;
```

```
                Reg3<= Reg2;
        end if;
end if;
end process;

Inst_fifo1:fifo port Map(
                                clk         => clk,
                                sinit=> rst,
                                din  => Reg3,
                                wr_en=> wr_fifo1,
                                rd_en=> rd_fifo1,
                                dout => dout1,
                                full => full1,
                                empty=> empty1,
                        data_count => fifo1_count
        );

process(clk,rst,sh2_reg)
begin
if rst ='1' then
                Reg4<=(others=>'0');
                Reg5<=(others=>'0');
                Reg6<=(others=>'0');
elsif clk'event and clk ='1' then
        if sh2_reg ='1' then
                Reg4<= dout1;
                Reg5<= Reg4;
                Reg6<= Reg5;
        end if;
end if;
end process;

 Inst_fifo1_rd_sh2: fifo1_rd_sh2 PORT MAP(
                clk                     => clk,
                rd_done         =>rd_done,
                rst                     => rst,
                fifo1_count     => fifo1_count,
                rd_fifo1        => rd_fifo1,
                sh2_done                => sh2_done,
                sh2_reg         => sh2_reg,
                wr_fifo2                => wr_fifo2
        );

 Inst_fifo2:fifo port Map(
                clk         => clk,
                sinit=> rst,
                din  => Reg6,
                wr_en=> wr_fifo2,
                rd_en=> rd_fifo2,
                dout => dout2,
                full => full2,
                empty=> empty2,
                data_count => fifo2_count
        );

process(clk,rst,sh3_reg)
begin
if rst ='1' then
                Reg7 <=(others=>'0');
                Reg8 <=(others=>'0');
                Reg9 <=(others=>'0');
elsif clk'event and clk ='1' then
```

```
        if sh3_reg ='1' then
                Reg7 <= dout2;
                Reg8 <= Reg7;
                Reg9 <= Reg8;
         end if;
end if;
end process;
        Inst_fifo2_rd_sh3: fifo2_rd_sh3 PORT MAP(
                clk                     =>      clk,
                rd_done         =>      start,
                rst                     =>      rst,
                fifo2_count     =>      fifo2_count,
                rd_fifo2        =>      rd_fifo2,
                sh3_reg         =>      sh3_reg ,
                sh3_done                =>      sh3_done
        );

        Inst_reg_mac: reg_mac PORT MAP(
                clk                     =>      clk,
                rst                     =>      rst,
                rd_done         =>      rd_done,
                sram_wr                 =>      sram_wr,
                COEF_REG1       =>  COEF_REG1,
                COEF_REG2       =>  COEF_REG2,
                COEF_REG3       =>  COEF_REG3,
                COEF_REG4       =>  COEF_REG4,
                COEF_REG5       =>  COEF_REG5,
                COEF_REG6       =>  COEF_REG6,
                COEF_REG7       =>  COEF_REG7,
                COEF_REG8       =>  COEF_REG8,
                COEF_REG9       =>  COEF_REG9,
                Reg1            =>      Reg1 ,
                Reg2            =>      Reg2 ,
                Reg3            =>      Reg3 ,
                Reg4            =>      Reg4 ,
                Reg5            =>      Reg5 ,
                Reg6            =>      Reg6 ,
                Reg7            =>      Reg7 ,
                Reg8            =>      Reg8 ,
                Reg9            =>      Reg9 ,
                Mac_out                 =>      data_out
        );

end Behavioral;
```

Appendix B

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

--Uncomment the following lines to use the declarations that are
--provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity main_16 is
   Port (       clk                         : in      std_logic;
        rst                         : in      std_logic;
                        dout_real           : out std_logic_vector(12      downto
        0);
                        dout_imag           : out std_logic_vector(12      downto
        0);
                        starting_data   : out std_logic;
                        Ending_data         : out std_logic
                    );

end main_16;

architecture Behavioral of main_16  is
type state is (start,count1_en_gen,count1_en_gen_hold0,
        count1_en_gen_hold1,count1_en_gen_hold2,count1_en_dis0,
        count1_en_dis1,dly,dly1,count1_en_dis2,dly2,dly3,dly0,dly4,dly6);
signal ps,ns:state;

signal counter1                 :       std_logic_vector(3     downto        0);
signal count_out1       :       std_logic_vector(3     downto        0);
signal counter2                 :       std_logic_vector(3     downto        0);
signal count_out2       :       std_logic_vector(3     downto        0);
signal counter3                 :       std_logic_vector(3     downto        0);
signal count_out3       :       std_logic_vector(3     downto        0);
signal count2                   :       std_logic_vector(8     downto        0);
signal address_s_1      :       std_logic_vector(3     downto        0);
signal address_s_2      :       std_logic_vector(3     downto        0);
signal wr_en_s                  :       std_logic;
signal done                     :       std_logic;

signal sel_mux,sel_mux_s    :       std_logic_vector(1     downto        0);
signal sel_mux_2,sel_mux_s_2 :      std_logic_vector(1     downto        0);
signal reg_real_0,reg_real_1,reg_real_2,reg_real_3  :       std_logic_vector(8
downto          0);
signal reg_imag_0,reg_imag_1,reg_imag_2,reg_imag_3:std_logic_vector(8
downto          0);
signal data_out_re_lut,data_out_im_lut:std_logic_vector(10
downto          0);
```

```
signal real_out_s_1,imag_out_s_1                                    :
        std_logic_vector(10    downto
0);
signal real_out_s_s,imag_out_s_s            :       std_logic_vector(10    downto
0);
signal Mux_real_out,Mux_imag_out :std_logic_vector(8    downto
0);
signal Mux_real_out_2,Mux_imag_out_2                                :
        std_logic_vector(10    downto
0);
signal imag_out_s_s_2,real_out_s_s_2                                :
        std_logic_vector(12    downto
0);
signal X0_real_s,X1_real_s,X2_real_s,X3_real_s          :       std_logic_vector(8
downto          0);
signal X0_Imag_s,X1_Imag_s,X2_Imag_s,X3_Imag_s          :       std_logic_vector(8
downto          0);
signal counter1_s
        :       std_logic_vector(3     downto          0);
signal counter2_s
        :       std_logic_vector(8     downto          0);
signal dina_real_s                                                  :
        std_logic_vector(8     downto          0);
signal dina_imag_s                                                  :
        std_logic_vector(8     downto          0);
signal we_a
                :       std_logic;
signal counter1_en                                                  :
        std_logic;
signal we_real,we_imag                                              :
        std_logic;
signal done_s
        :       std_logic;
signal count1_en,reg_en                                             :
        std_logic;
SIGNAL count2_en,wr_en,count3_en                                    :
        std_logic;
signal wr_en_2_s,WR_EN_2                                            :
        std_logic;

type state1 is (start,dly,stop);
signal ps1,ns1 : state1;

signal flag,flag_1                          :       std_logic_vector(1     downto
        0);
signal reg_real_3_2,reg_real_2_2 :  std_logic_vector(10    downto          0);
signal reg_real_1_2,reg_real_0_2    :       std_logic_vector(10    downto          0);
signal reg_imag_3_2,reg_imag_2_2  :         std_logic_vector(10    downto          0);
signal reg_imag_1_2,reg_imag_0_2 :          std_logic_vector(10    downto          0);
signal count_out3_2,count_out2_2 :  std_logic_vector(3     downto          0);
signal real_out_s_2,imag_out_s_2    :       std_logic_vector(12    downto          0);
```

```
signal data_out_imag_lut_2,data_out_real_lut_2                                  :
std_logic_vector(10   downto        0);
signal x3_real_s_2,x2_real_s_2,x1_real_s_2,x0_real_s_2          :
std_logic_vector(10   downto        0);
signal x0_imag_s_2,x1_imag_s_2,x2_imag_s_2,x3_imag_s_2          :
std_logic_vector(10   downto        0);
signal counter1_2,count_out1_2,counter2_2,counter3_2            :
std_logic_vector(3    downto        0);
signal count1_en_2,reg_en_2,count2_en_2,count3_en_2             :       std_logic;

signal count1_en_2_s        :       std_logic;
signal flag_s,flag_s_1      :       std_logic;

COMPONENT radix4_butterfly_10 PORT(
                X0_Real             : IN    std_logic_vector(8    downto        0);
                X0_Imag             : IN    std_logic_vector(8    downto        0);
                X1_Real             : IN    std_logic_vector(8    downto        0);
                X1_Imag             : IN    std_logic_vector(8    downto        0);
                X2_Real             : IN    std_logic_vector(8    downto        0);
                X2_Imag             : IN    std_logic_vector(8    downto        0);
                X3_Real             : IN    std_logic_vector(8    downto        0);
                X3_Imag             : IN    std_logic_vector(8    downto        0);
                W0_real             : IN    std_logic_vector(10 downto  0);
                W0_imag             : IN    std_logic_vector(10 downto  0);
                Clk                 : IN    std_logic;
                Rst                 : IN    std_logic;
                Real_Out            : OUT std_logic_vector(10   downto        0);
                Imag_Out            : OUT std_logic_vector(10   downto        0));
END COMPONENT;
COMPONENT rom_16_10 PORT(
          clk               : in    std_logic;
                rst                 : in    std_logic;
                address             : IN    std_logic_vector(3 downto 0);
                dout1       : OUT std_logic_vector(10 downto 0);
                dout2       : OUT std_logic_vector(10 downto 0));
END COMPONENT;
COMPONENT register1
        PORT(
                Reg_in              : IN    std_logic_vector(8 downto 0);
                Clk                 : IN    std_logic;
                Rst                 : IN    std_logic;
                reg_en      : IN    std_logic;
                Reg_out             : OUT std_logic_vector(8 downto 0));
END COMPONENT;
COMPONENT de_mux PORT(
                d_in                : IN    std_logic_vector(8 downto 0);
                sel                 : IN    std_logic_vector(1 downto 0);
                out1        : OUT std_logic_vector(8 downto 0);
                out2                : OUT std_logic_vector(8 downto 0);
                out3                : OUT std_logic_vector(8 downto 0);
```

```
                out4                    : OUT std_logic_vector(8 downto 0));
END COMPONENT;
component  ram_s  port (
        addr                            : IN    std_logic_VECTOR(3 downto 0);
        clk                             : IN    std_logic;
        din                             : IN    std_logic_VECTOR(8 downto 0);
        dout                            : OUT std_logic_VECTOR(8 downto 0);
        we                                      : IN    std_logic);
END component;

--================================================================

signal real_final_s,imag_final_s        : std_logic_vector(12 downto 0);
signal full_r_final,empty_r_final       : std_logic;
signal full_i_final,empty_i_final       : std_logic;
signal rd_en_final,rd_en_final_s        : std_logic;

type state3 is (start,rd_en_gen,stop);
signal ps3,ns3 : state3;

--================================================================
--===========FIRST STAGE COMPONENT
DECLARATION=======================

COMPONENT rom_2nd_stage PORT(
          clk                   : in    std_logic;
                rst                     : in    std_logic;
                address                 : IN    std_logic_vector(3 downto 0);
                dout1           : OUT std_logic_vector(10 downto 0);
                dout2           : OUT std_logic_vector(10 downto 0));
END COMPONENT;

--================================================================
--======SECOND STAGE COMPONENT
DECLARATION==========================

COMPONENT de_mux_2
        PORT(
                d_in                    : IN    std_logic_vector(10 downto 0);
                sel                     : IN    std_logic_vector(1 downto 0);
                out1                    : OUT std_logic_vector(10 downto 0);
                out2                    : OUT std_logic_vector(10 downto 0);
                out3                    : OUT std_logic_vector(10 downto 0);
                out4                    : OUT std_logic_vector(10 downto 0));
END COMPONENT;
COMPONENT radix4_for_2nd_stage
        PORT(
                X0_Real                 : IN std_logic_vector(10 downto 0);
                X0_Imag                 : IN std_logic_vector(10 downto 0);
                X1_Real                 : IN std_logic_vector(10 downto 0);
                X1_Imag                 : IN std_logic_vector(10 downto 0);
```

```
                X2_Real         : IN std_logic_vector(10 downto 0);
                X2_Imag         : IN std_logic_vector(10 downto 0);
                X3_Real         : IN std_logic_vector(10 downto 0);
                X3_Imag         : IN std_logic_vector(10 downto 0);
                W0_real         : IN std_logic_vector(10 downto 0);
                W0_imag         : IN std_logic_vector(10 downto 0);
                Clk             : IN std_logic;
                Rst             : IN std_logic;
                Real_Out    : OUT std_logic_vector(12 downto 0);
                Imag_Out    : OUT std_logic_vector(12 downto 0));
END COMPONENT;
COMPONENT register1_2 PORT(
                Reg_in          : IN std_logic_vector(10 downto 0);
                Clk             : IN std_logic;
                Rst             : IN std_logic;
                Reg_En          : IN std_logic;
                Reg_Out         : OUT std_logic_vector(10 downto 0));
END COMPONENT;

--================================================================
--=================FINAL STAGE SPRAM================================

component ram_s_2 port (
        addr                    : IN std_logic_VECTOR(3 downto 0);
        clk                     : IN std_logic;
        din                     : IN std_logic_VECTOR(10 downto 0);
        dout                    : OUT std_logic_VECTOR(10 downto 0);
        we                              : IN std_logic);
end component;
component fifo_final
        port (
        clk                     : IN std_logic;
        sinit                   : IN std_logic;
        din                     : IN std_logic_VECTOR(12 downto 0);
        wr_en                   : IN std_logic;
        rd_en                   : IN std_logic;
        dout                    : OUT std_logic_VECTOR(12 downto 0);
        full                    : OUT std_logic;
        empty                   : OUT std_logic);
end component;

type state2 is (start,rd_en_gen,count1_en_gen_hold0,
        count1_en_gen_hold1,count1_en_gen_hold2,count1_en_dis0,
        count1_en_dis1,dly,dly1,count1_en_dis2,dly2,dly3,dly0,dly4,dly6);
signal ps2,ns2:state2;

--====================== MAIN ARCHICTECTUTE BEGINS==================
begin
```