Method and Implementation of Multi-Channel Correlation in the Hybrid CPU+FPGA System

Maxim Leonov

A thesis submitted to
Auckland University of Technology
in partial fulfilment of the requirements for the degree of
Master of Engineering (ME)

2009

School of Engineering

Primary Supervisor: Dr. Slava Kitaev

Abstract

Modern high-performance digital signal processing (DSP) applications face constantly increasing performance requirements and are becoming increasingly challenging to develop and work with. In DSP paradigm, many researchers see potential in achieving algorithm speed-up by employing Field Programmable Gate Arrays (FPGAs) – reconfigurable hardware with parallelism feature. However, developing applications for FPGAs incur particular challenges on the development flow.

This work proposes a scalable hybrid DSP system for performing high-performance signal processing applications. The system employs a hybrid CPU+FPGA architecture of commercially available, off-the-shelf (COTS) FPGAs and central processing units (CPU) of personal computers.

In this work an example implementation of a multi-channel cross-correlator is investigated and delivered using a new development paradigm. The correlator is implemented on the XD1000 development system using a high-level FPGA programming tool – Impulse CoDeveloper. Analysis of DSP application development in a hybrid CPU+FPGA system employing the high-level programming tool Impulse C is presented. Potential of the selected tool to deliver algorithm speed-ups is investigated using reference multi-channel correlator software.

Particular attention is devoted to input/output (I/O) implementation, which is considered one of the most challenging problems in FPGA design development. This work delivers an I/O framework based on PCI Express interface for the proposed high-performance scalable DSP system. Using Stratix II GX PCI Express Development Board from Altera Corporation, a scalable and flexible communication approach for the multi-channel correlator is delivered. This framework can be adapted to perform other high-performance streaming DSP applications.

The outcomes of this work are a multi-channel correlator developed in a reconfigurable environment with new design methodology and I/O framework with software control application. The outcomes are used to demonstrate the potential of implementing DSP applications in a hybrid CPU+FPGA architecture and to discuss existing challenges and suggest possible solutions.

Acknowledgements

I would like to acknowledge the support and guidance of my supervisor, Dr. Slava Kitaev, whose patient advice and encouraging directions always came at the right time and without whom I would never have completed this work. I would also like to thank Dr. Hamid Gholam Hosseini for his valuable participation during the first part of the project.

I would also like to express my gratitude to Altera Corporation for providing discounts for hardware. Separately, I would like to thank an Altera support team representative, Steven, for providing invaluable help and adviceduring my experience with Altera hardware.

A separate gratitude is due to Impulse Accelerated Technologies for generously providing extended evaluation licence for their product, which allowed me to complete this project. Impulse C support team also deserves acknowledgment, without whose participation very little, if any at all, progress would have been made.

I would also like to thank XtremeData, Inc. for supplying the hardware with academic discount.

I thank Jungo Ltd. for providing evaluation licence for their software.

An appreciation goes also to Aidan Hotan from the University of Tasmania, who helped with initial experiments of this project.

Finally, I would like to thank my parents and friends for their patience and understanding during completion of this thesis.

Table of Contents

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma from a university or other institution of higher learning.

List of Abbreviations

ADC	Analog-to-Digital Converter
ASIC	Application-specific Integrated Circuits
CDMA	Code Division Multiple Access
CLB	Configurable Logic Block
COTS	Commercial, Off-The-Shelf
CPLD	Complex Programmable Logic Device
CPU	Central Processing Units
DAC	Digital-to-Analog Converter
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processing or Digital Signal Processor
ESL	Electronic System Level
FPGA	Field Programmable Gate Array
GPP	General-Purpose Processor
HDL	Hardware Description Language
HLL	High-Level Language
HLP	High-Level Programming
HMC	High-speed Mezzanine Connectors
HPC	High-Performance Computing
HSMC	High-Speed Mezzanine Connectors
I/O	Input/Output
JTAG	Joint Test Action Group
LE	Logic Element
MIPS	Million Instructions Per Second
MMAC	Million Multiply Accumulate Operations

MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
PAL	Programmable Array Logic
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCI-X	Peripheral Component Interconnect Extended
PSP	Platform Support Package
RADAR	RAdio Detection And Ranging
RAM	Random Access Memory
RC	Reconfigurable Computing
RF	Radio Frequency
RTL	Register Transfer Level
SAR	Synthetic Aperture Radar
SDRAM	Synchronous Dynamic Random Access Memory
SFP	Small Form-factor Pluggable
SOC	System On Chip
USB	Universal Serial Bus
UWB	Ultra-Wideband
VLSI	Very Large-Scale Integrated

List of Tables

List of Figures

List of Documents on CD-ROM

Root folder

Thesis final (Maxim Leonov).doc – thesis file in word format

Thesis final (Maxim Leonov).pdf – thesis file in PDF format

Folder \Impulse_C \ – correlator designs developed in Impulse CoDeveloper

Subfolder \6channel_correlator – six-channel correlator

\6channel_correlator\Correlator_C_hw.c – hardware source file

\6channel_correlator\Correlator_C_sw.c – software source file

\6channel_correlator\Correlator_C.h – include file

\6channel_correlator\input_signals.txt – file with input samples

\6channel_correlator\correlator_out.txt – file storing correlator's output

\6channel_correlator\Correlator_C.exe – generated file for desktop simulation

\6channel_correlator\Correlator_C.icProj – Impulse C project file

Subfolder \8channel_correlator – eight-channel correlator

\8channel_correlator\Correlator_C_hw.c – hardware source file

\8channel_correlator\Correlator_C_sw.c – software source file

\8channel_correlator\Correlator_C.h – include file

\8channel_correlator\input_signals.txt – file with input samples

\8channel_correlator\correlator_out.txt – file storing correlator's output

\8channel_correlator\Correlator_C.exe – generated file for desktop simulation

\8channel_correlator\Correlator_C.icProj – Impulse C project file

Subfolder \10channel_correlator – ten-channel correlator

\10channel_correlator\Correlator_C_hw.c – hardware source file

\10channel_correlator\Correlator_C_sw.c – software source file

\10channel_correlator\Correlator_C.h – include file

\10channel_correlator\input_signals.txt – file with input samples

\10channel_correlator\correlator_out.txt – file storing correlator's output

\10channel_correlator\Correlator_C.exe – generated file for desktop simulation

\10channel_correlator\Correlator_C.icProj – Impulse C project file

Subfolder \11channel_correlator – eleven-channel correlator

\11channel_correlator\Correlator_C_hw.c – hardware source file

\11channel_correlator\Correlator_C_sw.c – software source file

\11channel_correlator\Correlator_C.h – include file

\11channel_correlator\input_signals.txt – file with input samples

\11channel_correlator\correlator_out.txt – file storing correlator's output

\11channel_correlator\Correlator_C.exe – generated file for desktop simulation

\11channel_correlator\Correlator_C.icProj – Impulse C project file

Subfolder \12channel_correlator – twelve-channel correlator

\12channel_correlator\Correlator_C_hw.c – hardware source file

\12channel_correlator\Correlator_C_sw.c – software source file

\12channel_correlator\Correlator_C.h – include file

\12channel_correlator\input_signals.txt – file with input samples

\12channel_correlator\correlator_out.txt – file storing correlator's output

\12channel_correlator\Correlator_C.exe – generated file for desktop simulation

\12channel_correlator\Correlator_C.icProj – Impulse C project file

Subfolder \16channel_correlator – sixteen-channel correlator

\16channel_correlator\Correlator_C_hw.c – hardware source file

\16channel_correlator\Correlator_C_sw.c – software source file

\16channel_correlator\Correlator_C.h – include file

\16channel_correlator\input_signals.txt – file with input samples

\16channel_correlator\correlator_out.txt – file storing correlator's output

\16channel_correlator\Correlator_C.exe – generated file for desktop simulation

\16channel_correlator\Correlator_C.icProj – Impulse C project file

Folder \Correlation_reference_program – correlation reference program

\Correlation_reference_program\correlation.c – software correlation source code

\Correlation_reference_program\input_signals.txt – file with input samples

\Correlation_reference_program\results.txt – file storing correlator's output

Folder \I_O_framework – I/O framework design with integrated six-channel correlator developed in Impulse CoDeveloper

\I_O_framework\hw\ – folder with exported Impulse C correlation design

\I_O_framework\hw\lib\tx_ddr_resp.v – Tx DDR2 read response state machine (from reference design)

\I_O_framework\rx_pcie.v – Rx PCIe Receiver block (from reference design)

\I_O_framework\switch_top.v – switch module

\I_O_framework\ddr_ctrl_driver.v – correlator's DDR controller driver

\I_O_framework\pcie_ddr.stp – SignalTap Analyzer file

\I_O_framework\pcie_ddr.qpf – Quartus II project file

Folder \Sine_wave_generation – generation of model signals for correlation

\Sine_wave_generation\sine_generation.m – MATALAB script for signal generation

Folder \Software_Control_Application – software control operation for I/O framework

Software_Control_Application\altera_diag\altera\pci_dev_kit\diag\altera_diag.c – source file of software control application

Folder \Trial_hardware_correlator_design – trial hardware correlator design

\Trial_hardware_correlator_design\Correlator.vhd – VHDL top-level correlator entity

\Trial_hardware_correlator_design\CorrelatorLag.bdf – schematics of correlator lag

\Trial_hardware_correlator_design\FF.vhd – latch megafunction

\Trial_hardware_correlator_design\Correlator.qpf – Quartus II project file

\Trial_hardware_correlator_design\Correlator_model.mdl – Simulink model

\Trial_hardware_correlator_design\signal.mat – MATLAB data file with input signals

CHAPTER 1

Introduction

Necessity, who is the mother of invention.
—Plato

This chapter provides an overview of a high-performance DSP applications field from its origins to its current state. Appropriate background of the area of investigation is introduced and respective research objectives are outlined. The chapter concludes with contributions and organisations of this thesis.

1.1 Background

High-performance digital signal processing is very challenging work in today's engineering fields. Many applications face increasing performance demands and constant additional functional requirements. With digital signal processing becoming an integral part of everyday life, the demand for high-performance processing means has expanded rapidly in recent years.

Originally, signals in devices were manipulated using analog techniques (continuous-time domain). However, nowadays most of them are implemented in digital form (discrete-time domain). The genesis of the digital signal processing techniques can be connected to the advances in mathematical fields: finite difference methods, numerical integration method and numerical interpolation methods dating back to the seventeenth century. Of course, one of the major developments of the DSP area started in the 1950s, as a part of the far broader and embryonic field of digital computers. From the late 1960s, digital signal processing moulded into a separate field by itself. Thus, in the late 1970s when LSI (large-scale integration) technology became developed enough the realisation of a single chip DSP became practical (Mlynek, 1999). In 1978 AMI announced a "Signal Processing Peripheral" and released S2811 (Nicholson, Blasco, & Reddy, 1978) – a co-processor for a host micro. It was followed by Intel's 2920 in 1979 (Hoff & Townsend, 1979). The unique feature of the latter device was the on-chip analog-to-digital and digital-to-analog converters (ADC and DAC respectively), though

it lacked a multiplier. The DSP industry continued to grow and progress and, in the early 1980s, the world saw a second generation of DSPs with realised features like concurrency, multiple buses and on-chip memory. These were added with on-chip floating point operations in the third generation of DSPs in the early 1990s. In the late 1990s multi-processing features, image and video processors and low-power DSPs were introduced.

Contemporary signal processors are able to demonstrate much greater performance in many aspects: wider data buses and throughputs, higher processing speeds of up to 24,000 of 16-bit million multiply accumulate operations (MMACs) (Texas Instruments Inc., 2008), compatibility with various modern interfaces and buses such as PCI, USB, Ethernet and many others.

The means of performing signal processing are not, of course, limited to digital signal processors – the ever-growing field of signal processing invoked multiple solutions, architectures, technologies, tools and approaches: the major of which will be covered in the subsequent chapters of this work.

Many large-scale, high-performance DSP applications in such fields as radio astronomy, telecommunication, high-energy physics, and others involve computationally-intensive and therefore often time-consuming *correlation* of wideband signals. Correlation relies on the two most common types of composed DSP operations – multiply and accumulate (MAC), and multiply and add (MULT-ADD) operations. These operations have been implemented in digital processing successfully and efficiently. However, the challenge lies in the *number* of these operations, i.e. the *problem size* – the running time and/or space requirements of an algorithm. Many DSP applications employing correlation operation in their algorithms require real-time or near real-time processing, eg antenna aperture synthesis, medical applications, cellular and telecommunications applications (see 2.1 for details on these applications). Along with necessity to perform computations “on-the-fly”, correlation involves considerable execution time or *time complexity* for wideband correlation. For example, a multi-channel antenna array operating with 128 MHz bandwidth on each channel will yield a sampling rate of 256 MS/s with 8-bit sampling. For 8-channel correlation, this will produce 2 GB/s input data stream. Such correlation will generate 28 unique cross product outputs (the other 28 are just a mirror reflection of the first 28 (see 2.2.1)). An estimated number of operations required to perform a 32-lag correlation with these parameters is about 230 Giga-operations of real-time processing (Thompson, Moran, &

Swenson, 2001a). Being a classical DSP problem, correlation itself does not usually constitute a stand-alone, full and final application, rather it is an integral part of many DSP applications.

Here and throughout this thesis channel and antenna are used interchangeably. While such notation is acceptable and coomon for engineering and DSP fields, it differs in radio-astronmy where a channel is understood as a quantum of radio frequency bandwidth.

Although the processing capacity of DSP tools grew along with the requirements of the signal processing, the latter always outstands the former by considerable and everlasting margins. Almost as soon as the gap between ever-growing applications' requirements and capabilities of the DSP tools started shaping up (in the mid 1980s – see Figure 1.1) the search for counter-measures to close this gap started. The most prevalent and widely-used approach is *extensive* approach – gradual and proactive increase of the processing power of the DSP tools by increasing the number of employed computational units and/or operational parameters (operating frequencies, response times, storage capacities, etc.). Such approach proved to be productively working for Central Processing Units (or commonly known as processors), Digital Signal Processors (DSPs) and other conventional processing means for several decades and then started depleting quickly. The cost of the extensive approach hit the inevitable limitations very soon: high power consumption, complexity of dealing with growing number of computational units maintenance cost, etc.

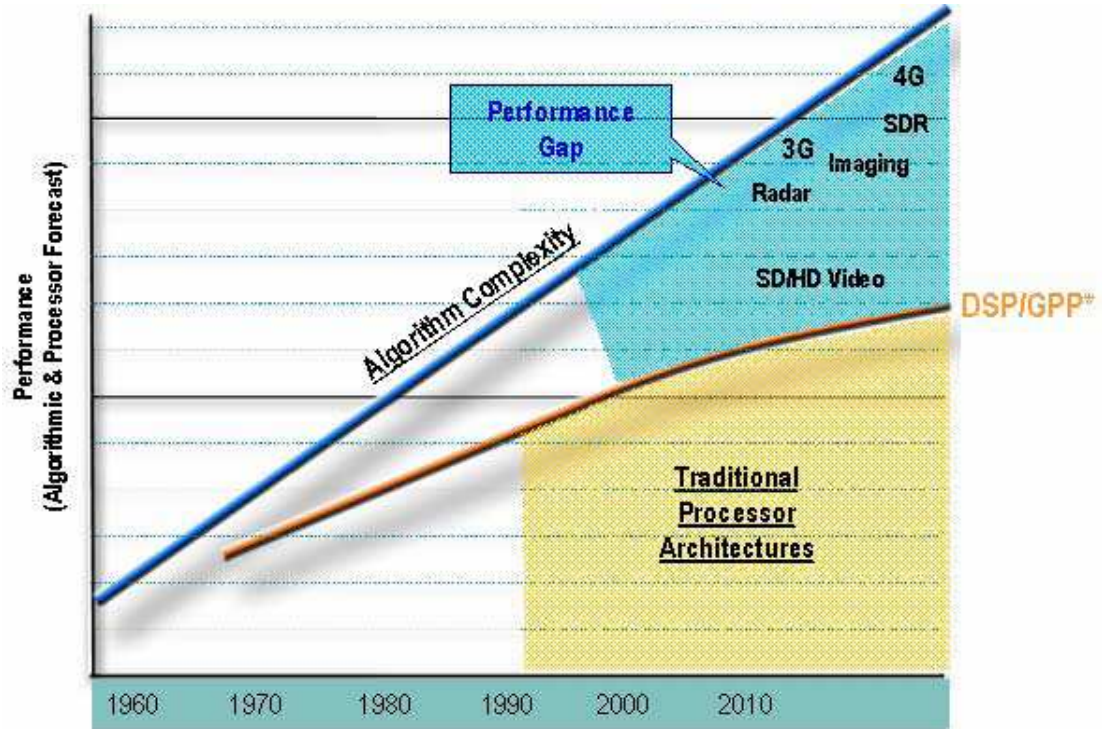


Figure 1.1. Performance Gap Between Traditional Processor Architectures and Growing Complexity of DSP Algorithms (Telikepalli & Fiset, 2006)

No surprise, that the research vector began deviating towards technologies and methods which could offer *intensive* ways of dealing with the problem as opposed to almost exhausted extensive approaches.

Currently an intensive approach is envisioned by many researchers in *parallelism* – simultaneous execution of several computational operations during one clock cycle. Moreover, parallelisation of applications is especially effective in the DSP field as long as many DSP algorithms possess intrinsic parallelism and therefore potentially sustain a large capacity for acceleration.

Application-Specific Integrated Circuits (ASICs) possess parallelism features and reach prominent efficiency of silicon utilisation for a specific operation determined during the manufacturing stage. Thus, they can be configured to meet the requirements of the particular application avoiding unnecessary generality. Reasonably, the performance efficiency achieved by ASICs for the targeted application is balanced by the impossibility of future modifications. Many ASICS' applications do not require any updates, modifications or alterations at all (eg integrated circuits of cell phones).

ASICS' counterparts – Field Programmable Gate Arrays also possess parallelism features. Along with this, FPGAs are *reconfigurable* devices, i.e. they can be reprogrammed in the field. Adding a bug fix, a new feature or even updating a

computational core of the application, significantly increases the flexibility of design. Due to this FPGAs have lower non-recurring engineering costs than ASICS. An FPGA is a semiconductor device consisting of programmable logic blocks and programmable interconnects. Along with parallelism, reconfigurability significantly expands the application scope of these devices. Armed with these two features over the years, FPGAs became one of the most promising technologies in digital electronics in general and in the DSP field in particular.

The interest towards FPGAs has risen even more radically in recent years with the growth of the chip capacities and the number of supported interfaces, which include, but are not limited to: PCI family interfaces, Ethernet family interfaces, support for memory interfaces like DDR/DDR2/DDR3/QDRII, USB, HyperTransport, RapidIO, VMEbus and many others. Moreover, FPGAs are particularly suitable for DSP applications due to: inherited parallelism in many DSP algorithms; high bandwidths to on-chip and external memories, which support multiple access ports thus allowing further exploitation of algorithm's parallelism; streaming to-be-processed data directly to computational core implemented in FPGAs via available high-speed interfaces. Hence, these FPGAs' features make them a very attractive option for applications' acceleration or even a competitive alternative for traditional DSP techniques, attracting more and more attention from the DSP industry.

With widespread availability of commercially available FPGAs in the late 1980s, the term reconfigurable computing (RC) was introduced. A reconfigurable computing system is a system which is built from reconfigurable computing devices, eg FPGAs or FPGA-like devices. These systems have to be reprogrammable, permit orders of magnitude speed-ups versus traditional computational systems and support hardware-like levels of performance (Guccione, 2008).

However, developing DSP applications for RC systems contain many more challenges and complexities than implementing applications in traditional software programming domain of DSPs, CPUs, etc. One of the main reasons is that FPGA design flow adheres to hardware development flow, which traditionally deals with low-level hardware description languages and demands explicit configuration of available resources in FPGA. The following issues also impose substantial challenges when employing reconfigurable hardware in traditional DSP applications: hardware state ambiguity complicates design debugging; parallelism consideration and a "run-at-a-clock" concept impose certain idiosyncrasies on algorithm implementation; explicit

memory structure puts constraints on storing of design variables. Moreover, conventional processing methods (DSPs, CPU and related) have been employed in the signal processing field considerably longer than FPGAs and therefore have more advanced and powerful developing and debugging tools.

Therefore, employing FPGAs either as a computational accelerator or as a stand-alone DSP application platform can be beneficial and challenging at the same time. Nowadays the traditional approach of increasing the processing capacity of computational means diverge from the traditional approach of raising the number of employed semiconductors (Moore's law) and operating frequency to a multicore and parallel execution approach. Many DSP algorithms possess intrinsic parallelism. FPGAs are a very attractive and potentially beneficial option to be employed in DSP paradigms for processing acceleration. Compelling reported speed-ups of 10X to 100X (Gokhale & Graham, 2005) of equivalent software algorithms attract more and more attention from the DSP community. Another argument to employ FPGAs for DSP algorithms is that these devices follow the International Technology Roadmap for Semiconductors (ITRS) (http://www.itrs.net/) even more narrowly than modern microprocessors (eg in terms of contained SRAM memory or leading on the first fabrication lines). Many researchers agree on a high potential of simultaneous operation of conventional processing unit(s) such as a CPU of a PC and reconfigurable hardware such as FPGA (Andrews et al., 2004; Milrod, 2006; Tahernia, 2005). This architecture invokes previously unavailable possibilities and options in signal processing but it also brings new challenges in development flow.

In this thesis a new design methodology for developing applications in a hybrid CPU+FPGA environment is applied. Using a mixture of traditional hardware development tools and conventional software development tools, a multi-channel wideband cross-correlation for DSP application on a hybrid CPU+FPGA architecture will be implemented. The prime objective of this thesis is to investigate the capabilities and challenges of this reconfigurable, hybrid architecture in the DSP field.

1.2 Research Objectives

In this work, we will investigate the implementation of a classical DSP problem – wideband multi-channel cross-correlation in a hybrid environment of a commercial, off-the-shelf CPU and FPGA. There are a number of contributions contained within this thesis.

First, the given work addresses the problem of computational deficiency in DSP field. By implementing a classical DSP problem in a hybrid CPU+FPGA architecture, its abilities of achieving speed-ups for applications from DSP fields are argued.

The second contribution is the platform and the workflow for developing high-performance DSP applications in a hybrid CPU+FPGA environment. The development workflow applied in this work is different from a traditional hardware design methodology. Rather than using low-level HDLs for hardware design implementation, the given work utilises high-level languages (HLLs) for hardware configuration. The potential of using HLLs for FPGA designs is evaluated and discussed. The given work delivers valuable outcomes for any DSP engineer developing applications in reconfigurable hardware with the aid of high-level programming (HLP) languages.

The third contribution is that this work tackles one of the most crucial issues of DSP applications, which becomes especially challenging and difficult in the FPGA domain – input/output interfaces. The I/O framework developed in this work features original method of interfacing to FPGA via onboard SDRAM simultaneously with high-speed communication with PC via PCIe interfaces. The developed method can be beneficial to many applications requiring extensive data exchange. Particularly, it can be useful for applications targeting Altera's PCIe Development Kit Stratix II GX Edition or to any Altera's devices featuring PCIe and DDR/DDR2 SDRAM interfaces.

The given work also introduces the possible evolution of the proposed platform. The number of available interfaces on the exploited FPGA board (PCIe, Ethernet, SFP, HSMC, etc) and simple connectivity options of the conventional PC box provide a considerable degree of architectural possibilities. A highly scalable platform for high-performance signal processing is proposed as a potential future development of the created design (section 3.2). In addition, one of the advantages of the suggested platform is the affordable cost as compared to proprietary DSP solutions: the cost of the prospective system is composed merely from FPGA board's and PC box prices.

1.3 Thesis Layout

The thesis is organised into seven chapters. Chapter 2 briefly introduces the background of the investigated problem. Computationally intensive DSP applications employing cross-correlation of signals are discussed. Cross-correlation theory is discussed, which is followed by a discussion on DSP implementation technologies.

In Chapter 3 the notion of a hybrid CPU+FPGA architecture is introduced. Challenges existing in this architecture and high-level programming of reconfigurable hardware are discussed. The chapter also proposes high-performance DSP hybrid architecture.

Chapter 4 discusses methodology applied in this work and introduces five stages of the full project design flow. These stages define the hardware and software development tools used at every particular stage.

Chapter 5 presents implementation flow of the project. Implementation details of the stages are defined in the previous chapter.

Results and outcomes of project implementations are presented in Chapter 6. They are discussed in Chapter 7. Approaches and solutions to alleviate known shortcomings and challenges of the developed project outputs, along with future developments are suggested.

CHAPTER 2

Theory Background and Related Work

We live in a moment of history where change is so speeded up that we begin to see the present only when it is already disappearing.
—R. D. Laing

This chapter will discuss the most common digital signal processing applications in a high-performance domain. First, a brief outline of generic digital signal processing algorithm will be given. Then, the next section will highlight the most common high-performance DSP applications, which will be followed by the discussion on the cross-correlation problem as integral and often one of the most computationally intensive parts of these applications. The remaining section will present and consider contemporary DSP implementation technologies.

The term digital signal processing implies converting an analog signal into a form of numbers (digital form), the processing of the resultant sequences to either obtain information or to synthesise signals with desirable properties and possibly convert the output into analog form again. The overall scheme of the generic DSP algorithm is shown in Figure 2.1:

Figure 2.1. Generic Digital Processing Scheme (Mitra, 2006)

The high-performance DSP applications feature a considerable amount of computations in the "Digital processor" stage. Several typical high-performance DSP applications are considered in the following section.

2.1 Typical High-Performance Signal Processing Applications

The number of signal processing applications in today's life is truly enormous. Nevertheless, not every DSP application is suitable for reconfigurable computing. A number of studies exist which investigate efficiency criteria of an application to be

employed in FPGAs. The application's performance implemented in FPGA depends on (Hutchings & Nelson, 2008):

1. Data parallelism available in the application's algorithm;
2. Data element size and arithmetic complexity;
3. Amenability to pipelining, and simple control requirements.

The following sections highlight several high-performance DSP applications, which have one common and integral operation – cross-correlation of signals. The potential of implementing these applications in or with the help of reconfigurable hardware will be considered. The applications below will be considered in retrospect.

2.1.1 Radio Astronomy

Throughout human history, man has been always mysteriously attracted to the sky. With the discovery and subsequent invasion of new technologies, traditional methods of visual investigation of the sky, ie methods of optical astronomy, were joined by *radio astronomy* techniques. Many astronomical bodies emit radio waves, which after certain processing can tell valuable and previously inaccessible information about their origin. Thus, in the last half of the 20^{th} century the prominent advances in radio astronomy led to a number of foremost discoveries like masers, pulsars, radio galaxies, the Cosmic Microwave Background Radiation, etc.

With radio astronomy, scientists can study astronomical phenomena which are invisible to the human eye. While in optical observation the useful information is extracted from the spatial distribution of light across an object, ie image, radio astronomy uses a different principle. RF waves emitted by a certain phenomenon can be received and directly sampled in a time domain, thus the tools used for detecting and measuring this interaction are considerably different from optical telescopes (Carroll & Ostlie, 2007).

To produce a radio image of a celestial phenomenon a principle of *interferometry* is used, which entails the superposition technique - interference (adding or overlaying) of signals from two or more antennas. This technique is also known as *antenna aperture synthesis* when multiple antennas are used to work as one using interferometry principle.

The core idea of antenna aperture synthesis is again to superimpose the signal waves from a number of radio telescopes and, while doing so, inphase waves will add

up, while antiphase waves will cancel each other out. This creates a combined telescope with the size of the furthest observing telescopes apart. The image quality produced by such a composed antenna depends on the number of the projected separations between any two telescopes as seen from the radio source (number of *baselines*). With each radio telescope producing a data stream the processing and computational task can be extremely intensive. Besides, the processing is complicated by low signal-to-noise ratios which are common for radio astronomical observations.

The backbone operation of antenna aperture synthesis is correlation or finding the amount of similarity in the signal between two given antennas in an antenna array. The term correlation and underlying theory will be discussed more deeply in 2.2.1. Even for a medium-size antenna array, computation of correlation between all the elements of the array can be a very challenging task due to the number of involved mathematical calculations. The example considered in 1.1 with an 8-element antenna array requires 230×10^9 operations per second. An experienced reader will estimate that the problem size of the given example as average to below average. Nevertheless, such a system might require a performance power of not less than ~230 GFlops (depending on implementation). In real-life, large-scale systems that correlate signal pairs of multi-element arrays may contain millions of correlator circuits in order to accommodate all the required antennas and spectral channels. Hence, with an increase of any of the above parameters the computational complexity of aperture synthesis grows drastically. That is why antenna aperture synthesis and radio astronomy have been established as one of the most major and demanding consumers of DSP technologies.

2.1.2 RADAR Applications

RADAR or Synthetic Aperture Radar (SAR) applications are based on the principle of the scattering of electromagnetic waves. Originally, RADAR meant RAdio Detection And Ranging, however later the term became used as a standard word. The most common RADAR system consists of a transmitter and a receiver – EM waves radiated by the transmitter are reflected (scattered) by a target, which are then collected by the receiver for further analysis. Any change in the dielectric constants of the target and a media surrounding it will be conveyed in the scattered waves. The basic principle of RADAR theory is illustrated in Figure 2.2.

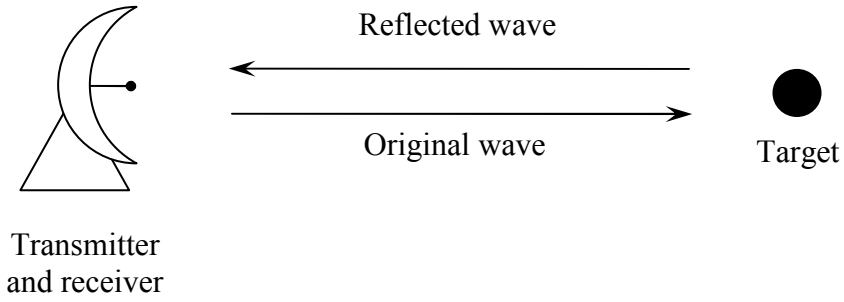


Figure 2.2. Basic RADAR Principle

The gathered data can include the object's position, movement or its particular features and attributes. The range of the applications, where RADAR technique is used, is wide: weather prediction, air traffic control, threat detection systems, military missile guidance and reconnaissance radars, etc.

Antenna aperture technique as mentioned in 2.1.1 is also used in phased array radars. In such arrays, comprised of a number of similar properly displaced antenna elements, the scanning beam is controlled by operating a phase of the signal of each individual transmitting antenna. Thus, the overall transmitted signal is maximised in a desired direction and suppressed in undesired directions.

In modern RADAR applications DSP techniques are used extensively: generation and forming of the transmission pulses, controlling the antenna beam's pattern and direction, filtering of clutter, and beamforming (S.Bhaktavatsala, 2002). Cross-correlation is one of the central operations in RADAR applications: it is used to find the relation or similarity between the original and reflected waves. When applied on a largescale for multiple signals and performed in a real-time fashion, such correlation becomes a challenging task.

Substantial utilisation of RADAR techniques in military area lays particular requirements on the DSP technologies in RADAR applications, eg a common trend is the need for smaller energy-efficient systems with high processing capabilities. Furthermore, typical operational signals in RADAR are very weak and with the recent tendency of radars being operated in a dense urban environment, the task of processing such signals becomes a major challenge. This issue can be mitigated by "overlaying data from multiple sensors and known terrain features". In addition, newly-emerging digital beam-forming technologies based on a high-speed digital systems work with an ever-increasing number of scanning beams. The latter two issues increase RADAR system processing requirements considerably (Kenny, 2007).

2.1.3 Medical Applications

Nonetheless, signal-processing technologies are not solely used for cognitive purposes. Perceptive, non-intrusive analytical capabilities of radio imaging make it an excellent option for diagnoses in *medical area*s. In the context of digital signal processing, the most interesting category amongst all the categories comprising the medical imaging is the ultra-wideband (UWB) imaging, which in turn is used primarily for early-stage breast cancer detection.

One of the most crucial factors in successful breast-cancer treating is detecting it at the earliest stage possible. Contemporary diagnosis methods like X-ray imaging, Magnetic Resonance Imaging (MRI), and ultrasound are capable of reducing malignant tissue. However, problems with a relatively high rate of false-negative diagnosis (Huynh, Jarolimek, & Daye, 1998) along with many unnecessary biopsies due to the low positive predictive rate (Elmore et al., 1998) make the use of X-ray mammography difficult and ineffective. Other methods like MRI and ultrasound are somewhat more effective in cancer lesions detection, yet do not always provide the necessary level of sensitivity, can be too operator specific and are very expensive.

Many of the existing drawbacks in early-stage breast-cancer detection can be alleviated with ultra-wideband imaging technology. The UWB imaging method employs the radar technique which was described in 2.1.2. A transmitting antenna (or a set of antennas) radiates a scanning burst of microwave energy. This electromagnetic energy penetrates through the target under investigation, scatters on the target, and further is collected by a receiving antenna or with an array of antennas. Then, the processing takes place with the primary goal to identify the presence and location of the considerable backscattered energy – an indication of the dielectric difference between malignant and healthy tissue. Thus, the post-processing of the received signals has to be very sensitive to filter out the necessary information from the antennas' noise, clutter due to heterogeneity in the breast tissue etc. Moreover, it has to be precise – image resolution on the order of millimetres is desirable (Li, Bond, Veen, & Hagness, 2005). Similar to RADAR applications, during the post-processing stage correlation is applied to find similarities or discrepancies in tissue readings provided by transmitted and reflected waves. Therefore, modern approaches of the existing radar application have to be adapted and improved, according to the requirements of UWB medical imaging. Now the medical diagnosis tools are still expect the DSP instruments to deliver an efficient and reliable method of breast-cancer testing.

2.1.4 Telecommunication

Communication technologies are one of the most actively developing areas today. The emergence of new wireless services along with the high growth of data rates in existing services, indicates an ever-growing demand for telecommunication capacity. With worldwide deployment of 3G networks, releasing beyond-3G and 4G standards and specifications, the challenges for the DSP area keep accumulating: high data throughputs (up to 1 Gbps), multimedia communications, seamless global roaming, maintaining high user capacity, and supporting migration and the compatibility between existing previous-generation and upcoming next-generation networks, etc (Ibnkahla, 2004). Consequently, the research community is focusing on different advanced signal processing issues to achieve substantial improvements in communication systems.

To demonstrate the computation requirements that lay in the telecommunication area, an example of Code Division Multiple Access (CDMA) standards can be used. CDMA based standards (CDMA2000, W-CDMA, etc.) have become increasingly popular during the emergence of the third generation networks due to their objective to maintain the ever-growing data throughputs and efficient spectrum utilisation. In brief, the idea of CDMA implies that a number of users share the same bandwidth of frequencies and are distinguished by the individual code (pseudorandom code). Such an approach has a much higher data bandwidth than traditional Time and Frequency Division Multiple Accesses (TDMA and FDMA respectively). However, these benefits are balanced with certain difficulties. For instance, the choice and assignment of a pseudorandom code to user is not a very simple routine in highly populated large-scale mobile networks. This problem can be computationally-intensive so certain solutions were proposed to address this issue (B.-J. Chang, 2007). Similarly, an analogous problem arises on the receiving side – to decode signal from multiple users in the most efficient and fastest way. It has been indicated that this problem also has significant computational needs (Agarwal, B.V.R.Reddy, & K.K.Aggarwal, 2006).

Furthermore, the underlying complexity of the CDMA algorithm implies a challenging and complicated processing mission itself: as long as in CDMA the users share the same bandwidth the *multiple access interference* (MAI) has to be considered and alleviated. Prevention of this interference is exacerbated by the *intersymbol interference* (ISI) and multipath signal propagation which is natural to all urban mobile networks. For this purpose sophisticated *channel estimation* algorithms are applied. The computational complexity of such algorithms is considerable and furthermore they have

to be implemented in real-time fashion. Therefore, the research community has turned to elaborate DSP techniques like real-time DSPs and FPGAs to respond to these challenges (Ouameur & Massicotte, 2007).

Tackling the processing difficulties is not of course the feature of only the CDMA standard. As it was mentioned above, with the rapidly-growing rates, throughputs, capacities, etc. the industry is facing expanding requirements throughout its applications. One example is 3G and 4G mobile standards. These standards offer high data throughputs to the end-users – even comparable to office LANs' in 4G networks. To supply such high speeds, a number of advanced and complex techniques are employed in these standards. One such technique is smart antennas. To maintain high data rates in complex urban environments these antennas use adaptive beamforming and direction-of-arrival (DOA) estimation algorithms. In turn, these algorithms employ cross-correlation operation for estimation which signals arriving from which directions to suppress and which to maximise. Such calculations have to be performed with complex numbers and most importantly should be done in real-time. Thus, it is evidently seen that the necessity for high-performance signal-processing utilities spans across the whole communication industry, leaving researchers in unrelenting pursuit for an adequate response.

2.2 Correlation as a Typical DSP Application Problem

In many of the aforementioned applications an integral and common part can be singled out – all of them are dealing with combined sources of information providing a synergistic combination of knowledge about the investigated object. In other words, whenever a system is dealing with a number of input data streams collaboratively reducing the entropy of a studied phenomenon, the term "multi-sensor data fusion" is applied (Stergiopoulos, 2000). The integral part of this fusion is to express the joint result of analysis of two or more originally different sources. For that reason a *correlation* operation is applied, which in turn is regarded no less as a "backbone" of the whole DSP area.

Thus, the prevailing number of high-end DSP applications such as antenna aperture synthesis, radioimaging, RADAR, radio astronomy, high-energy physics and many others, has a common *and* very computationally-intensive part – the multi-channel wideband correlation of signals. Correlation or, more generally speaking, finding a relation between a set of signals, is a computational core for the majority of

signal processing operations and is considerably critical for computation performance. The result of the cross-correlation function is "a measure of similarity between a pair of energy signals" (Mitra, 2006).

As it was noted before in 2.1.1, one of the applications where correlation is applied is *radio astronomy*. For example, it is used in the radio-astronomical technique known as Very Long Baseline Interferometry (VLBI). In turn, the antenna aperture synthesis is used in VLBI. The latter technique implies that the correlated product of signals from two radiotelescopes gives *visibility frequencies* of celestial object. The frequency information is obtained by averaging additional multiplications by a lagged signal and finally the data is transferred to the frequency domain by applying Fourier transform (Thompson, Moran, & Swenson, 2001b).

2.2.1 Correlation Theory

A measure of similarity between a pair of signals, $x[k]$ and $y[k]$, is given by the *cross-correlation* $r_{xy}[k]$ sequence:

$$r_{xy}[n] = \sum_{k} x[k]y[k-n] \tag{2.1}$$

where the lag index $n \in [-N/2, N/2 - 1]$, k is the time index, N is a number of lags and typically is a power of two. The *lag* term denotes the time-shift between the pair of signals with negative ($n < 0$) and positive ($n \geq 0$) lags being distinguished. Basically, the number of lags defines how many points or output values the correlation produces. In real life applications, where for example the correlation function is used together with Fourier transform, the number of lags can be referred as the *resolution* of correlation. A device which performs correlation of a set of signals is called a *correlator*. The number of lags is an important characteristic of a correlator along with the *number of channels*, ie number of supported input signals. When a signal is correlated with itself, such an operation is called *autocorrelation* and is often used in filtering and other processes.

One should note the incurred execution time or *time complexity* for wideband correlation. For a wideband signal according to the Nyquist condition, the processing involves computation of a greater amount of samples, hence the processing duration increases. In addition, the results of correlation computation abide to the following law (Thompson et al., 2001b):

$$N = \frac{N_S(N_S - 1)}{2} \tag{2.2}$$

where N is the total number of the cross-products and N_S is the total number of antennas (sources) to be correlated. Hence, wideband multi-channel correlation embraces a considerable amount of computations.

Strictly speaking, Equation (2.2) gives the number of *unique* correlation results or half of the total correlation results – the remaining half can be obtained by simply reversing the results from the first half. The latter issue is caused by the following property of correlation: correlation of $x[k]$ with $y[k]$ is not the same as correlation of $y[k]$ and $x[k]$. So, putting down mathematical notation of correlation of $y[k]$ with $x[k]$:

$$r_{yx}[n] = \sum_{k=-\infty}^{\infty} x[k]y[k-n] = \sum_{l=-\infty}^{\infty} y[l+n]x[l] = r_{xy}[-n] \tag{2.3}$$

Thus, $r_{yx}[n]$ is obtained by time-reversing sequence $r_{xy}[n]$.

2.2.2 Digital Correlators

As mentioned above, the number of lags is an important feature defining the resolution capabilities of a correlator. The higher the number of lags, the better a correlator can "tell" how similar two signals are to each other. In reality, the number of lags is set by the application's requirements and defines the number of multiply-and-accumulate and multiply-and-add computations. The latter statement is true for *digital correlators*, ie correlators that work with a stream of digitised samples $x[n]$ from an analog output $x(t)$. Two general types of digital correlators are distinguished:

- Lag or XF Correlator
- FX Correlator

In the lag or XF correlator Fourier transform to the frequency domain is performed *after* cross multiplication of signals. The number of channels in such correlators is an integral power of two with the signals' bandwidths also divisible by two to be compatible with digital computing techniques (Thompson et al., 2001a).

Whereas in the FX correlator Fourier transform is performed *before* cross multiplication of signals. Therefore, the total number of operations on the FX correlator is proportional to the number of antennas or more correctly signals coming from these

antennas, whereas in the XF correlator the amount of computation is proportional to the number of antenna (signal) pairs. Hence, the FX correlators are more economical in terms of hardware requirements especially for a considerable number of signals (Thompson et al., 2001b).

2.2.3 Implementations of Correlators

Correlators can be implemented in hardware or software. Normally, hardware correlators are designed and manufactured for a certain and specific application and are implemented in Very Large-Scale Integrated (VLSI) circuits. The CABB Hardware Correlator (Ferris, 2006) is an example of a hardware correlator. This correlator has a complex and very large-scale architecture comprising a number of VLSIs, multiplexers, accumulators, filter banks and other devices. It is utilized in Australia Telescope Compact Array to process signals from six 22 m. antennas of Australia Telescope Compact Array. In addition, FPGAs are used in this correlator as well – to produce different configurations of filter banks.

As for software correlators, they are implemented as a set of libraries or computer programs to perform the designated task: correlation of a given set of signals. Amongst known and acknowledged software correlators the following need to be mentioned:

K5 Software Correlator (Imai, Koyama, & Kondo, 2005) is probably one of the most famous correlators implemented in software. Currently the K5 correlator is involved in the VERA project in Japan and furthermore in collaborative work of Korea and Japan in the project “East Asian Correlator” in Seoul (Kawaguchi, Kobayashi, & Oyama, 2006). K5 is an FX correlator.

Swinburne University of Technology has another software-based correlator. Initially this correlator was XF-type (West, 2004) but it was considered slow and the recently new FX correlator DiFX has been implemented and tested (A. B T. Deller, Tingay, Bailes, & West, 2007). Both correlators have been implemented on the Linux parallel high-performance parallel cluster utilizing the Message Passing Interface (MPI) standard for process-to-process communications. There was a reported intention to explore hybrid architecture (ie comprising FPGA and Swinburne cluster) within this FX correlator (A. Deller, 2005).

The Jet Propulsion Laboratory of California Institute of Technology designed Softc software correlator (Lowe, 2004). Launched as one of the many test programs to replace an outdated hardware correlator Block I in the Delta-Differenced One-way

Range (DeltaDOR) spacecraft navigation system, Softc underwent a lot of changes and finally was employed in Mars Odyssey, Mars Exploration Rover, Deep Space 1 and other missions. It has significant processing accuracy (not less than 10^{-13}); it can correlate 1, 2, 4, and 8-bit sampled data, upper, lower, or double sideband data and data using one of either two encoding schemes.

Software correlators are known for their flexibility and possible high spectral resolution along with broad bandwidth (A. B T. Deller et al., 2007). This is achieved by employing high-performance computer systems, eg clusters, massive-parallel computing systems (MPCS or MPC), etc. A cluster is an interconnected group of computers working together as a single computer. The backbone of the clusters is high performance computing units, ie *nodes*. Contemporary clusters involved in high-end digital signal processing applications are considerably complex and elaborate systems with multiple-level architectures and high-speed interconnects. MPC systems are computer systems that include multiple independent processing units running in parallel. Examples of MPC computers include Blue Gene and Earth Simulator amongst others.

The international TOP500 list encompasses the 500 fastest and most powerful computing systems around the world (www.top500.org). As of November 2007, the top supercomputer is the Department of Energy's IBM BlueGene/L system in USA with a performance of nearly 500 TFlops. Another BlueGene/L computer located at the University of Groningen performs correlation tasks in a Low Frequency ARay (LOFAR) project. It consists of 12,288 700 MHz dual PowerPC 440 cores yielding 34.4 TFlop/s of correlation performance (Romein, Broekema, Meijeren, Schaaf, & Zwart, 2006a).

Nevertheless, such performance comes at a price – development time and maintenance cost balance this substantial computational power. With BlueGene/L's power consumption of 27.6 kW per rack (IBM Corporation, 2006) the LOFAR's six-rack supercomputer consumes 165.6 kW per hour. Besides, the estimated development time is one man-year (Romein, Broekema, Meijeren, Schaaf, & Zwart, 2006b). One of the reported issues with the LOFAR's correlators is the lack of the high bandwidth in BlueGene crucial for streaming DSP applications and overall necessity of faster intercommunication between the cores. Moreover, software correlators require substantial debugging and testing of the code: "eliminating of processing errors and inaccuracies" was one of the "greatest hurdles" in Softc correlator implementation

(Lowe, 2004). Developing, debugging and testing can be generalised as one of the greatest hurdles for all high-performance DSP systems.

Along with the considerable complexity of developing high-performance DSP systems go their relevant energy requirements. At the Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08), Pete Beckman from Argonne National Laboratory in his keynote speech made a strong point about the power consumption requirements of contemporary and future supercomputers (Beckman, 2008). In particular, it was predicted that within years the power consumption of a computational system would become the most determinative characteristic. In time, increasing the processing capabilities by increasing the operational frequency and adding additional transistors (Moore's law), depleted itself and gradually diverted to multi-core and parallel execution of the algorithms, where currently most of the research and development work is carried out. In turn, the same is envisioned for parallel operation – parallelisation of the applications and algorithms will eventually exhaust with Flops per Watt ratio becoming the systems' performance measuring unit. Therefore, with power requirements becoming one of the most significant factors additional constraints are laid upon the development of high-performance DSP systems and, most importantly, on the technologies applied in these systems.

2.3 DSP Technologies

2.3.1 The Performance Requirements of the DSP Applications

The number of mathematical calculations involved in the aforementioned high-end DSP applications is extremely high. For instance, to perform only a 1,024-point FFT yields 10,240 complex multiplications and additions per operational cycle. Moreover, to provide trustworthy data, a radio telescope observing a celestial phenomenon has to employ FFT with even higher resolution as well as a number of other operations, eg correlation of wideband radio-frequency signals, thus yielding even higher number of computations. On top of that, any DSP application, whether it is an image processing routine or telecommunication operation, demands these computations be executed in a rapid manner.

Besides, relentlessly expanding requirements of today's electronic systems keep pushing the resources contemporary DSP instrumentation towards and over the verge of

depletion. Figure 2.3 illustrates the performance gap that has emerged in the communication industry between increasing algorithm complexity originated from recent "standards revolution" and existing processing architectures.

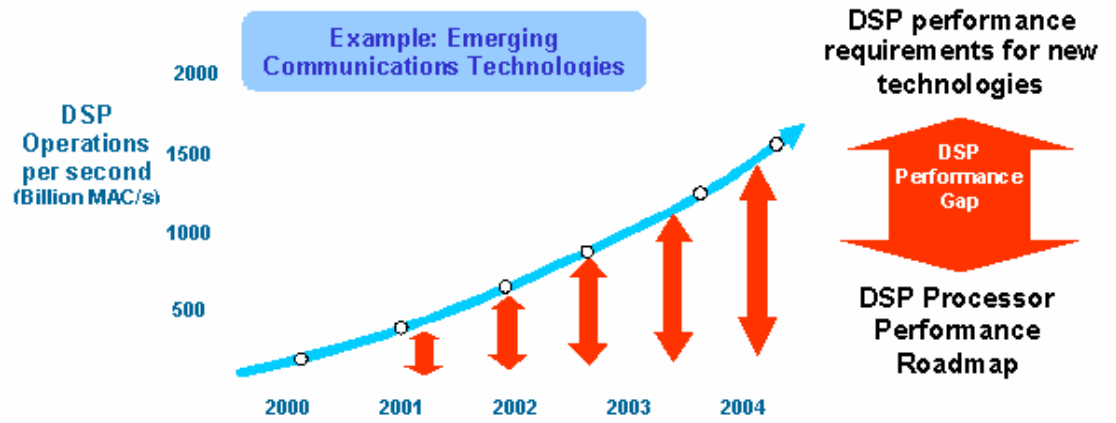


Figure 2.3. The DSP Performance Gap in Communications Industry (Ganousis, 2004)

So, how can one approach the ever-growing demands of the DSP field? The most universal approach to meet the substantial and constantly growing requirements of high-end DSP applications is to increase the processing power of computational units (CPUs, DSPs, ASICs, etc).

This has a number of limitations and drawbacks such as:

- High power consumption, which in turn leads to necessity of efficient power dissipation;
- Complexity of accomodating a large number of transistors in a single chip, which are growing with each year according to Moore's Law;
- High market costs.

Hence, this is not always feasible to cover the requirements of a certain high-end DSP application by simply involving more computational power (units) due to the hardware constraints in contemporary tools. Therefore, the search focus has to be shifted towards renovating or enhancing the existing apparatuses or creating new ones. The next sections cover the most common tools available in the DSP field.

2.3.2 Digital Signal Processors (DSPs)

Currently there is a number of tools in the DSP area. One of the major tools for DSP applications are Digital Signal Processors. DSPs were first created in the late 1970s – S2811 (Nicholson et al., 1978) and Intel's 2920 (Hoff & Townsend, 1979). Although

Intel's device did not have a multiplier, it already had on-chip ADC and DAC – a feature still present in the modern DSPs. The 1980s saw a second generation of DSPs with supported concurrency, multiple buses and on-chip memory. These were added with on-chip floating point operations in the third generation of DSPs in the early 1990s. In the late 1990s multi-processing features, image and video processors and low-power DSPs were introduced.

Today DSPs are produced by semiconductor vendors such as Texas Instruments, Analog Devices, Motorola and others. Contemporary top-level DSPs are capable of achieving substantial speeds – for example the high-performance multi-core TMS320C6474 from Texas Instruments can achieve up to 24,000 million instructions per second (MIPS) or 24,000 16-bit MMACs per cycle (Texas Instruments Inc., 2008). This DSP is also equipped with a 16/32-bit DDR2-667 Memory Controller, EDMA3 Controller, 1000 Mbps Ethernet MAC interface, two 1x Serial RapidIO Links and many other peripheries.

In general, DSPs are a specialized form of microprocessor designed specifically for digital signal processing. Nowadays DSPs have a well-developed tool set – typically a high-level programming language as C++. DSPs perform real-time processing and have fixed hardware architecture with certain set of resources. Hence, DSPs have reconfigurability freedom only to the extent of the programming code running on them. Furthermore, the performance requirements of today's DSP applications have now exceeded the capabilities of even such powerful DSPs as Texas Instruments' TMS320C6474.

Another common platform for performing DSP applications – Application-Specific Integrated Circuits (ASICs) possesses an alternative approach for performing signal processing applications.

2.3.3 Application-Specific Integrated Circuits (ASICs)

The inception of Application-Specific Integrated Circuits or more commonly ASICs started in 1980s when the now-defunct Ferranti Company released the first gate-array – Uncommitted Logic Array (ULA). The first Uncommitted Logic Arrays contained only a few thousand gate circuits (transistors, logic gates, and other active devices) and they did not perform any specified function. A particular function of a ULA was configured by adding a final layer of metal interconnects to the ULA thus connecting the elements on the ULA in the desired, customised fashion. The later versions of these early

developments became more complicated with a greater number of gates and in some cases included RAM elements.

Modern ASICs retain the same ideology – they perform only limited sets of tasks laid in them during manufacturing stage. These devices are capable of performing their limited sets of functions faster than general-purpose DSPs. Due to application-specific circuitry ASICs are able to employ high-speed functions of the targeted algorithm in the optimized hardware (Kuo & Lee, 2001). Most commonly ASICs are used for implementing well-tested and well-defined algorithms, eg Reed-Solomon coders in digital subscriber loop (xDSL) modems or stack functionalities of CDMA2000 standard in cell phones.

Depending on the grade of flexibility, three levels of ASICs are distinguished:

- Gate Array is the least customisable. Transistors, gates and other devices are predefined but unconnected – no metallization layers exist. A user specifies interconnection between the elements thus defining the function of the device. Today these devices are gradually replaced by structured ASICs where many features are predefined by the manufacturer: IP cores, power and clock sources, etc. This significantly reduces the design time, as a user has to specify much fewer design technicalities.
- Standard cell methodology has a high degree of flexibility. It assumes that the ASIC's design is defined by a user from the cell libraries created by the manufacturer and , therefore, has much less space for mistake than full custom design.
- Full custom design is the most flexible and, therefore, the most expensive and time-consuming approach. It assumes developing an ASIC from transistor level.

Despite that ASICs can perform their specified application faster than general-purpose DSPs, they do posses their own challenges and limitations. The most obvious limitation of ASICs originates from their most prominent strength: hardware optimised for performing dedicated applications means little or, most often, absolutely no degree of algorithm flexibility.

Another challenge with ASICs is that they are configured with hardware description languages (HDL) such as Verilog, VHDL and some other less popular options. These languages are low-level programming languages and differ significantly

from high-level programming languages employed for programming conventional general-purpose DSPs. The challenges of hardware description languages are more broadly discussed in section 3.1.2.

Single DSP or ASIC can be employed as a platform for single or several signal-processing applications. In the case of large-scale high-performance DSP applications, they may be employed as building blocks in sizeable computational systems such as supercomputers, computational clusters, grid computing, etc.

2.3.4 High-Performance Computing

For performing large-scale DSP applications, high-performance computing (HPC) systems can be used. HPC systems (supercomputers or computer clusters) comprising multiple computational processors communicate through versatile types of interconnect. The types of DSP applications employed on HPC systems are exceedingly large-scale and include but are not limited to: correlation of wideband RF signals involved in radio observation of celestial objects (eg CABB (Ferris, 2006) or DiFX (A. B T. Deller et al., 2007) Australian correlators), video-centric applications of new generation wireless telecommunications standards, such as wireless videoconferencing, real-time video streaming, etc. (Gentile & Wills, 2004) and many others.

Over the years, the HPC proved to be an effective and sophisticated tool for performing DSP applications. Technologies and tools applied in HPC have significantly developed over the past years – density of transistors on processors (Moore's law), communication speeds and throughputs, number of processors performing one task, uniform memory access with few or no caches, etc. In addition, modern HPC systems are practically linearly scalable.

Moreover, HPC systems have the potential to perform the assigned task *in parallel*, ie the task is split into several parts, each of which is performed by a separate computational unit in parallel (Wilkinson & Allen, 2004). This is achieved by either using multiple computational processors within a single computer, ie a multiprocessor, or by multiple computers working on a single problem. The possibility to perform tasks in parallel becomes radically beneficial for DSP applications as most of them can be easily *parallelised*. More precisely the majority of DSP applications fall under the Single Instruction, Multiple Data streams (SIMD) category in taxonomy introduced by Michael J. Flynn (Flynn, 1972). Figure 2.4 illustrates SIMD architecture.

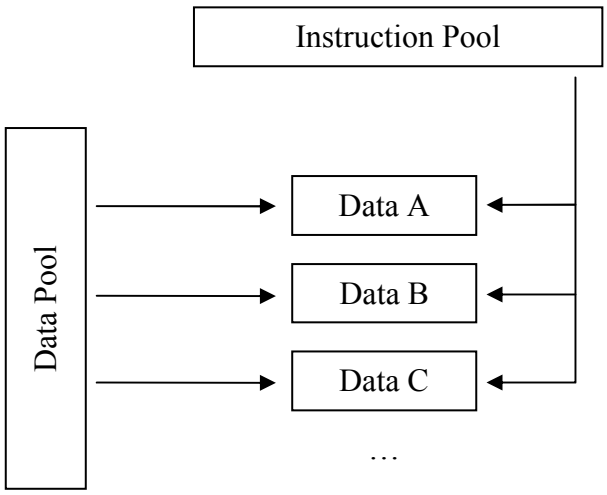


Figure 2.4. Single Instruction, Multiple Data (SIMD) model

In the SIMD model the same set of operations from the Instruction Pool is applied on different data streams Data A, Data B, etc from the Data Pool simultaneously and, therefore, this processing can be naturally parallelised.

A certain application can benefit from SIMD implementation if it involves a large number of the same repetitive operations applied on a large number of data bits. Many DSP applications satisfy this condition. For example, the correlation described by the Equation (2.1) consists of a number of simple mathematical operations, namely multiplication and addition which are applied to the same data set – samples of input signals. Hence, computation of a single sample of a correlation function involves a precisely calculated number of calculation routines on a certain input sample. These routines can be successfully parallelised thus attaining a speed-up in the performance which consecutively leads to power conservation and increased throughput.

Real-life SIMD implementation examples include Intel's MMX processors, their AMDs counterparts – 3DNow! Processors, Graphics Processing Units of PC video cards, and many others. The SIMD model is applied in large-scale supercomputers as well.

2.3.5 FPGAs as a DSP Tool

Another prominent tool for parallelisation is a maturing field of FPGAs, which has drawn massive attention in recent years from leading electronics developing vendors and designers throughout the world. Recent profound advances in the Field Programmable Gate Array area demonstrate that signal, image and video processing applications which are typically implemented on FPGAs, comprise complicated

calculations over a large amount of streaming data. These applications can gain substantial speed-up from available on-chip parallelism (Guo, Najjar, Vahid, & Vissers, 2004). The technological background of FPGAs is discussed in more details in 3.1.1.

An FPGA is a semiconductor device containing programmable logic blocks which can be interconnected and configured to meet the desired functionality specified by a certain application. Once an FPGA is programmed it operates as optimised hardware developed for a particular task. Designs incorporating FPGAs have at least two significant advantages in comparison with DSP devices and ASICs:

- Parallelism – the ability to perform several operations in parallel and therefore performs faster;
- Reconfigurability or, in other words, the ability to be customised for a certain application.

FPGAs' parallelism feature allows them to perform more operations at a single clock cycle than their conventional processing counterparts. Therefore, FPGAs operate at much lower frequencies than their conventional processing counterparts while achieving similar or even greater performance results. Lower operational frequencies lead in turn to lower power consumption, which has become one of the most crucial issues in recent years and is predicted to play an ever more dominant role in the foreseeable future of high-performance computational systems (Beckman, 2008). In-depth background of FPGAs is given in 3.1.1.

In the past years, the computational capabilities of commercially available FPGAs even overcame some commercially available CPUs in terms of achievable performance – see Figure 2.5.

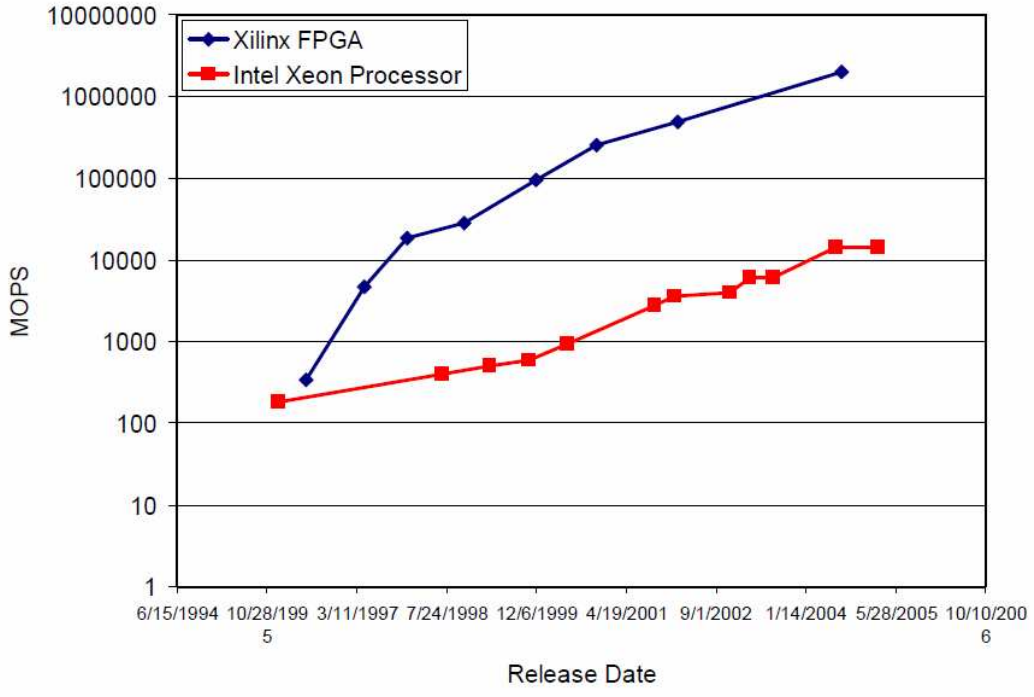


Figure 2.5. Moore's Law in the CPU and FPGA World (Chen Chang, 2005)

Figure 2.5 demonstrates millions of floating point operations per second (MOPS) achievable by the FPGA representative (Xilinx FPGA) and the CPU representative (Intel Xeon CPU) throughout their release dates. FPGA field is already renowned as a new computational paradigm (Ekas, 2007; Phillips, Littlefield, Dahlgren, & Ciufo, 2007) by the research community.

Nevertheless, along with prominent beneficial features FPGAs have certain challenges and drawbacks. First of all FPGAs are configured with low-level hardware programming languages which incur considerable programming and debugging efforts. Furthermore, all of the interfaces and features present on FPGAs have to be explicitly configured for each particular application/design. This and other challenges in FPGA programming discussed in 3.1.2, complicate the utilisation of FPGAs for DSP applications.

2.3.6 CPU+FPGA Hybrid Approach

Several approaches exist to overcome the challenges of FPGA programming. One is to employ high-level software programming tools and languages for hardware programming. This approach will be discussed in more detail in 3.1.3. Utilising software high-level programming languages for FPGA designs development intrinsically links to an approach which is sometimes called hybrid CPU+FPGA architecture.

More and more researchers express their interest towards a mutual operation of commodity computational means (eg CPUs, DSPs and even HPC servers) along with reconfigurable hardware (FPGAs). Some regard it is as an "optimal solution" (Milrod, 2006) and others merely acknowledge its persuasive benefits along with intrinsic challenges (Andrews et al., 2004; Tahernia, 2005). The CPU+FPGA approach may ease the tedium of designing FPGA applications by bridging the gap between more familiar software development tools and challenging hardware development tools of FPGAs. Besides,in CPU+FPGA architecture a design can employ the benefits of both conventional processing methods and optimised hardware implementation. A number of DSP applications can effectively employ this hybrid and flexible architecture. The next chapter introduces and discusses CPU+FPGA architecture in more detail.

2.4 Chapter Summary

This chapter introduces the background of the investigated problem. The following typical computationally-intensive DSP applications employing cross-correlation of signals are discussed: radio astronomy, RADAR applications, medical applications, and telecommunication.

The theory of the targeted cross-correlation problem with the focus on digital implementation of correlation is described. This is followed by a discussion on the computational requirements of the modern DSP applications. Traditional technologies (DSPs and ASICs) for performing DSP applications are considered along with methods of implementation of large-scale DSP applications (HPC). Their features and existing challenges are discussed.

Further, FPGAs as a potential tool to achieve performance acceleration for DSP algorithms are discussed. The CPU+FPGA hybrid approach is introduced.

CHAPTER 3

Hybrid CPU+FPGA Architecture

The future is always beginning now.
—Mark Strand

This chapter outlines the investigated CPU+FPGA architecture and highlights its potential advantages and challenges. Employment of this architecture for high-performance DSP applications is discussed along with the applicability of contemporary hybrid reconfigurable computing systems for performing these applications.

There are a number of solutions on the market when FPGAs are deployed inside a computer system, eg Cray XD1, SGI RASC, Nallatech H100 family blades for IBM BladeCenters, XtremeData, and SRC Computers with their proprietary MAP reconfigurable processor architecture. This option is of particular interest in this work since such architecture has a computational power of a general-purpose processor (GPP), along with an FPGA's flexibility of reconfigurable hardware, and, therefore, the possibility for performance acceleration of DSP applications through parallelisation.

3.1 Hybrid CPU+FPGA Architecture

The hybrid technology implies simultaneous work of an FPGA chip and a CPU of a commodity PC in one system. It might be particularly advantageous for such DSP applications such as antenna aperture synthesis, radio imaging, RADAR, radio astronomy, high-energy physics etc. A common and very computationally-intensive part in the above-mentioned applications is the multi-channel wideband correlation of signals. Such correlation can be implemented in a parallelised manner in an FPGA. Depending on the type of correlation (XF or FX) (Thompson et al., 2001b) both floating and fixed point numbers can be successfully and efficiently targeted to work on this architecture involving either a CPU or an FPGA as required.

In addition, reasonably decreasing prices of FPGA devices and the off-the-shelf availability of hardware architecture, place the CPU+FPGA approach as a promising alternative to the large-scale and high-cost correlators such as CABB Correlator (Ferris,

2006) or various software correlators (A. B T. Deller et al., 2007; Kawaguchi et al., 2006; West, 2004). If the interface between a CPU and an FPGA is established and has low latency, architecture can offer a flexible and powerful platform (Andrews et al., 2004; Milrod, 2006). CPU+FPGA architecture can also be more convenient as the CPU can be utilised to work with un-parallelisable tasks (fetching and streaming data samples into an FPGA, acquisition of correlated data, etc.) whereas an FPGA can be utilised for actual correlations (multiplication and accumulation operations).

3.1.1 FPGA Technology

Inception of FPGAs dates back to 1960s when Gerald Estrin's group at the University of California at Los Angeles did one of the first works on reconfigurable computing (Estrin, 1960, 2002). In 1984 Ross Freeman, co-founder of Xilinx Corporation invented a new type of semiconductor device which is now known as the Field Programmable Gate Array (Xilinx Inc., 1984).

FPGAs are historically connected to complex programmable logic devices (CPLDs). Figure 3.1 demonstrates that they belong to the same group called field-programmable logic (FPL):

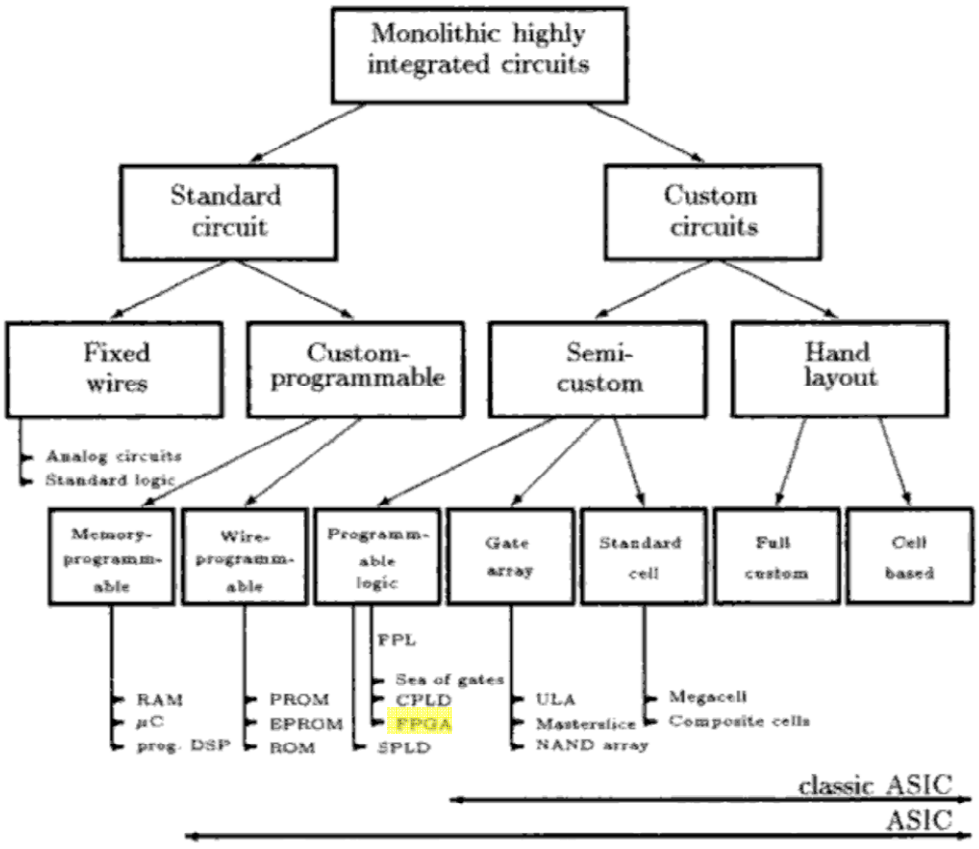


Figure 3.1. Classification of VLSI Circuits (Meyer-Baese, 2004a)

The structure of an FPGA is an evenly-spaced two-dimensional array tiled with logic blocks – Configurable Logic Blocks (CLBs). Each CLB represents a simple memory used as a lookup table and flip-flops for buffering. CLBs communicate with other logic blocks via a programmable interconnection network – see Figure 3.2.

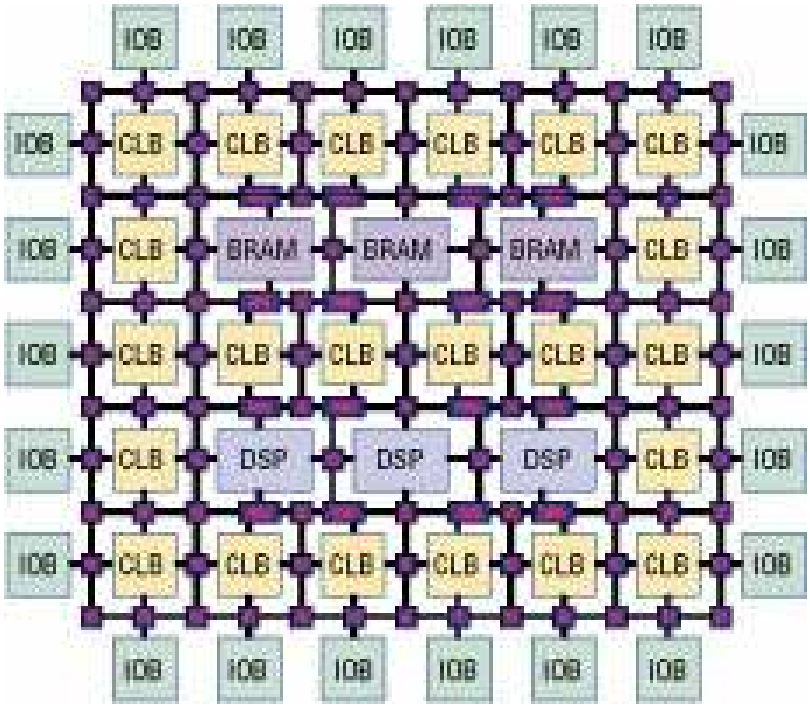


Figure 3.2. FPGA Internal Structure (Buell, El-Ghazawi, Gaj, & Kindratenko, 2007)

The peripheral blocks of an FPGA are I/O blocks (IOB in Figure 3.2) dedicated for communication between internal logic blocks and the I/O pins. Modern FPGAs' architecture features on-chip memory blocks as well as dedicated circuitry to perform DSP operations - DSP blocks.

The difference between FPGAs and said CPLDs lies in the *granularity* of a device, which designates the level of complexity of completing the routing between the blocks. Thus, FPGAs fall in the medium granularity devices group while CPLDs in the large granularity devices group. This distinction comes from the fact that CPLDs comprise simple programmable logic devices (simple PLDs or SPLDs) with common densities of several thousand to tens of thousands of logic gates, whereas FPGAs normally contain tens of thousands to several millions of logic gates.

In order to define the behaviour of an FPGA it needs to be programmed with a configuration bit stream first. These bit streams are generated from structural register transfer level (RTL) specifications expressed by a user in the form of the HDL descriptions – most commonly Verilog or VHDL. These HDL descriptions are created

by hardware designers and follow the design flow demonstrated in Figure 3.3 before the configuration stream is created.

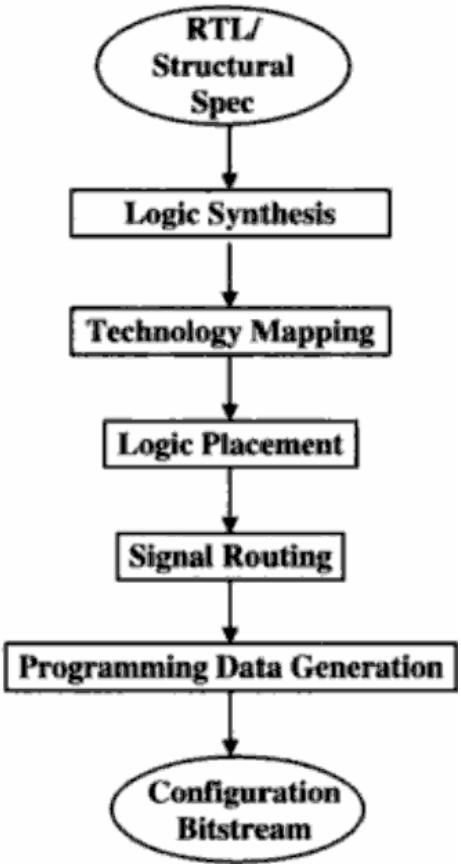


Figure 3.3. Low-level FPGA Design Flow (Gokhale & Graham, 2005)

The "Logic Synthesis" stage translates design descriptions in Verilog or VHDL into an optimised gate level representation. Along with Verilog and VHDL, hardware design descriptions can be done in schematics (Betz, Rose, & Marquardt, 1999). FPGA manufacturers supply their development tools with a number of predefined functions – intellectual property (IP) cores to simplify the development of complex FPGA designs. For example, Altera features such IP Cores (MegaCore functions) as PCI Express Compiler to create PCIe interface on the boards featuring PCIe connector, FFT, DDR Controller and others. During "Technology Mapping" stage, design primitives are converted into the netlist of physical resources on a selected FPGA chip. "Logic Placement" and "Signal Routing" stages are often combined in the literature and referred to as the "Place-and-Route" phase. This stage calculates and performs the most effective placement and interconnection of each mapped logic block on the specified device. Then, the last stage generates programming bitstream, which will configure various resources as required. Normally, the aforementioned stages are executed by the

proprietary tools from FPGA vendors. The results of each stage can be verified by the designer by means of timing analysis and simulation. A number of simulation tools are available from various vendors, eg ModelSim from Mentor Graphics.

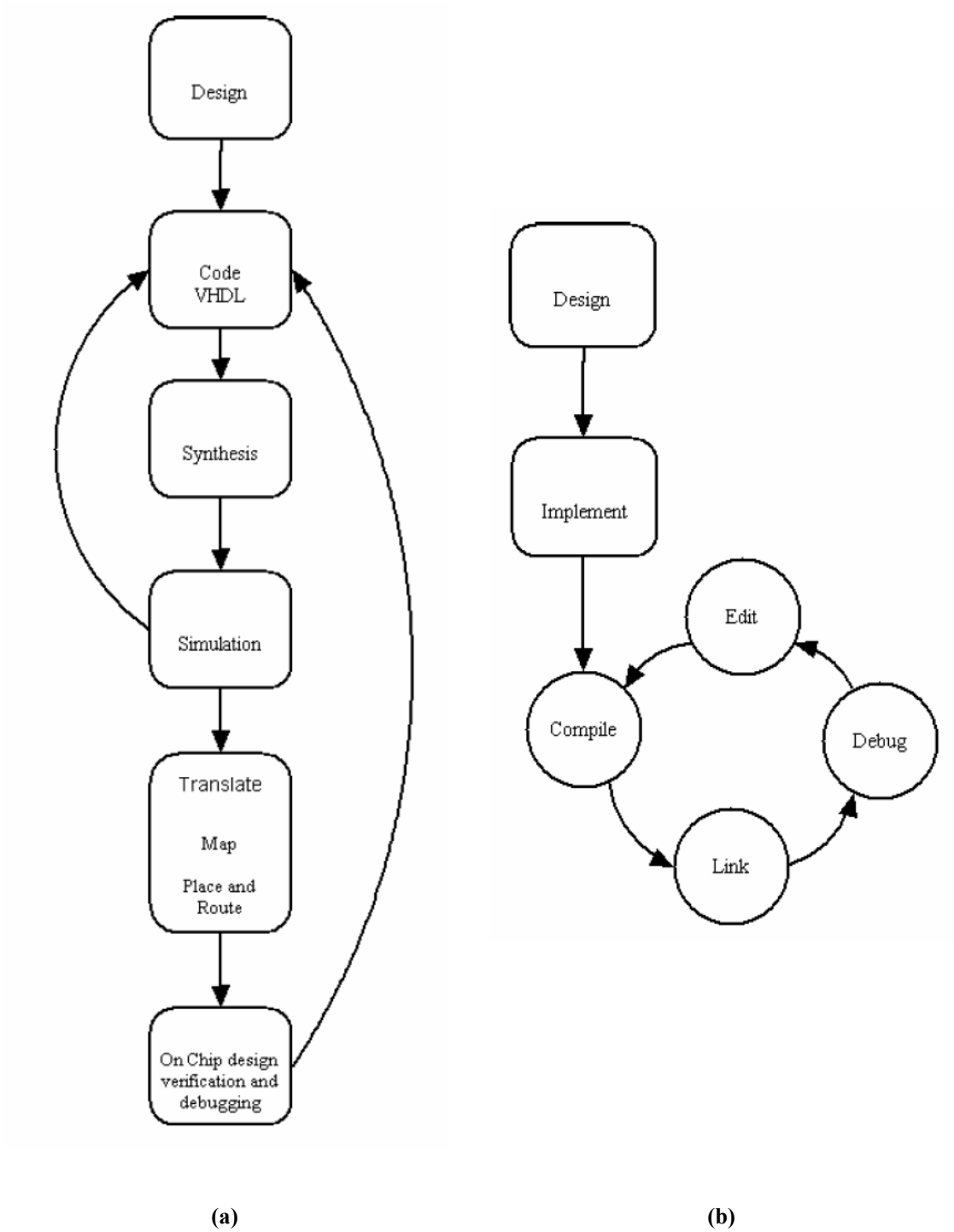
3.1.2 Challenges in FPGA Programming

Nevertheless, the very flexibility that makes FPGAs so universal and beneficial at the same time imposes a considerable challenge on the whole RC design process (Andrews et al., 2004; Tahernia, 2005; Wain et al., 2006). The following are the most prominent of the challenges that need mentioning:

1. A priori unawareness of FPGA about its I/Os. This issue implies that an FPGA initially knows nothing about how to communicate with external world. Any interface featured on an FPGA board has to be instantiated and configured in low-level specifications. To mitigate this FPGA vendors provide IP cores for most common interfaces. Robust, high-speed and low-latency I/O interfaces are a crucial component in the DSP paradigm (Milrod, 2006).
2. Compilation process and compilation time. Unlike conventional software programming where compilation normally takes seconds to minutes, hardware compilation is a complex task (see Figure 3.3) and may take hours to complete.
3. Storing variables in explicit memory hierarchy. In HDL each program variable has to be stored in the chosen memory type: external memory, on-chip memory, logic blocks configured as memory or registers. Changing the type of the selected memory might cause changes throughout the whole design (Gokhale & Graham, 2005).
4. Implicit hardware state in FPGA and complicated debugging. Debugging of the hardware design has to be carried out at the granularity of nanoseconds which is complicated by the lack of transparency of the hardware's state on FPGA (Gokhale & Graham, 2005)
5. Significant difference in hardware design flow and conventional software design flow. This issue is discussed in more detail in the next section.

3.1.3 High-Level Programming for Hybrid Architectures

The overall complexity of FPGA programming has been extensively studied in recent years and a number of solutions have been developed. Hardware and software design flows are considerably different. The general case of software and hardware design flows is depicted in Figure 3.4.



(a) **(b)**

Figure 3.4 Hardware (a) and Software (b) Design Flows (Wain et al., 2006)

A significant difference in the depicted flows is that software developers have a certain *level of abstraction* from the developed product. For example, virtual memory, hardware, cache, etc are determined by a processor's architecture in software flow, whereas in hardware flow these low-level design parameters have to be explicitly configured in every design. Ultimately, the hybrid CPU+FPGA architecture should envision transparent work with CPU and available FPGA resources as a seamless and integral computational system (Andrews et al., 2004). The mentioned level of abstraction is located on top of the RTL - see Figure 3.5.

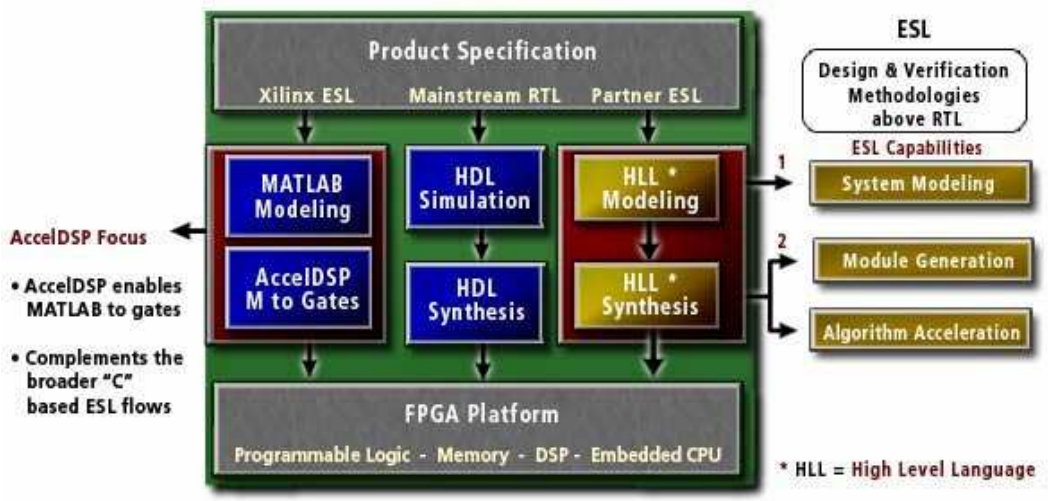


Figure 3.5. FPGA Electronic System Level (ESL) Approach (Xilinx Inc.)

The ESL Design Ecosystem approach shown in Figure 3.5 was developed by an international initiative launched by Xilinx Company, comprising a wide array of Ecosystem members. The original intention of this initiative is to deliver accessible and understandable tools for software designers, so they can develop hardware designs using traditional programming techniques.

It should be noted that C-based languages were never actually designed to employ parallelism in reconfigurable FPGA hardware. Therefore, each solution for high-level FPGA programming using sequential languages has to be able to employ algroeithm's parallelism and be aware of available resources of the targeted FPGA by means of libraries, support packages, etc (Baran, Bodenner, & Hanson, 2004; Wain et al., 2006).

During several years of activity, the ESL Design Ecosystem Initiative has developed a number of solutions: ImpulseC from Impulse Accelerated Technologies; Mitrion SDK from Mitrionics™, Inc; Cascade from CriticalBlue and many others. Many of the tools from and outside of the ESL Design Ecosystem Initiative were tested

and appraised for this work. The following criteria were used to determine the most applicable tool for this work:

1. Support of the selected hardware (see 4.2) or ability to generate HDL projects for non-specific platforms.
2. The tool's input must be high-level programming language (C, C++, C-like languages, etc).
3. Applied approach to extract parallelism: automatic, pragmas, manual adaptation of the input code, etc "Closeness" of implemented examples and tutorials to the selected application of multi-channel wideband correlation.
4. The tool has to produce synthesisable HDL code (Verilog or VHDL).
5. Evaluation option has to be present and the tool has to be affordable within the available budget (academia licence or alternatives).

Many of the tools were eliminated from consideration because they work only with a limited number of platforms, eg Mitron, Clarity from Mimosys, SystemCrafter SC, Reconfigurable Computing Toolbox r2.0 by DSPlogic, CoreFire by Annapolis Micro Systems etc. Some of the considered products like Sturbridge's Viva are merely graphical composition tools working with AND, OR, etc gates and logic operators, which can produce generic HDL codes. These type of tools do not deliver the necessary level of abstraction. Other tools were already near a defunct stage (Celxoica) or only at a really maturing stage (CHiMPS by Xilinx Research Labs).

DIMEtalk from Nallatech proved to be an interesting option. Although this tool primarily targets Nallatech's and Xilinx's boards, there is an option of generating HDL designs for generic platforms. However, the C to HDL conversion feature of DIMEtalk, which is positioned as additional and supplementary, was confirmed as insufficient for this project during evaluation of DIMEtalk product: DIMEtalk's approach of developing algorithms in high-level graphical interface was deemed too obscure and complex for selected application of multi-channel correlation.

Mitrion SDK (www.mitrionics.com) and DK Design Suite from Celoxica (www.celoxica.com) from the aforementioned instruments use pseudo-C languages such as Mitrion-C and Handel-C respectively. These languages explicitly express parallelism available in input design unlike the above-mentioned compilers and converters, which automatically seek for parallelism in the ingress code.

Catapult from Mentor Graphics is also a very promising tool, which unfortunately was not appraised since the manufacturer does not provide evaluation licences to academia.

Outside of Xilinx's ESL initiative are free Open Source projects from Los Alamos National Laboratory: Trident Compiler (Tripp, Peterson, Ahrens, Poznanovic, & Gokhale, 2005) and sc2 (Gokhale, Frigo, Ahrens, Popkin-Paine, & Stone, 2004) which both convert C or C++ code into synthesizable VHDL code and run under Linux operating system.

The XD1000 development system, which is used in this project, supports Impulse CoDeveloper from Impulse Accelerated Technologies and therefore it was selected as the main high-level FPGA programming tool. Selection of the XD1000 system and Impulse CoDeveloper tool highlights are given in sections 4.2.2 and 4.3.2 respectively.

3.1.4 Hybrid Systems

As it was mentioned before, many manufacturers eye the reconfigurable hardware as an integral part of the high-performance processing. Most of the top off-the-shelf manufacturers of high-end computing systems utilise FPGAs from other vendors and use them as small building blocks in their own solutions. The following solutions employ hybrid CPU+FPGA architecture:

- MAP processors for SRC-6 and SRC-7 systems from SRC Computers, Inc.
- XD development systems from XtremeData, Inc.
- Cray XR1 blade for XT5 system from Cray, Inc.
- SGI RC 100 blade (SGI RASC Technology) from Silicon Graphics, Inc.
- Nallatech H100 family blades for IBM BladeCenters from Nallatech, Inc.

A most interesting option in the context of this project is the XD family developments system from XtremeData.

Another vendor successfully employing both GPPs and FPGAs is XtremeData (www. xtremedatainc.com). At the very early stage, the company’s primary target was creating a fully-integrated Analytics Appliance for the Decision Support Systems applications. To sustain an intensive SQL query-processing characteristic for these applications, XtremeData came up with their primary IP component – FPGA-based In-

Socket Accelerators™. The efficiency of these accelerators was so compelling that the company began producing stand-alone FPGA-based In-Socket Accelerators. All of the XtremeData products (as of Autumn 2008) feature Altera-produced FPGAs – namely Stratix II family devices. The first generation accelerator – XD1000™ features one Stratix II EP2S180 device, whereas the second generation of accelerators – XD2000F™ and XD2000i™ devices (for AMD's Socket F architecture and for Intel FSB respectively) have two and three Stratix II FPGAs respectively. Such architecture with multiple FPGAs allows employing one of the chips as a communication bridge, while the rest perform actual application processing.

Most importantly, the XD1000 development system features the XD1000 Platform Support Package (PSP) for Impulse CoDeveloper, which enables full integration with Impulse C. This system will be used in this work and its exact role is given in 4.1.

However, having rigid and well-established I/O interfaces is one of the key aspects in DSP paradigm. Practically all of the aforementioned high-performance hybrid systems (eg XD1000, Cray XR1 etc.) lack data acquisition interfaces, thus making all their extensive computational powers and parallelisation capabilities unavailable for real-time high-performance DSP applications where high-speed I/O data interfaces are integral.

To overcome this shortcoming a prominent research group located at the University of California, Berkeley has developed the Berkeley Emulation Engine 2 or shortly BEE2 (C. Chang, Wawrzynek, & Brodersen, 2005). BEE2 is a high-end reconfigurable computer (HERC) consisting of computer modules connected through a global communication network. Figure 3.6 shows a block diagram of the BEE 2 computer module.

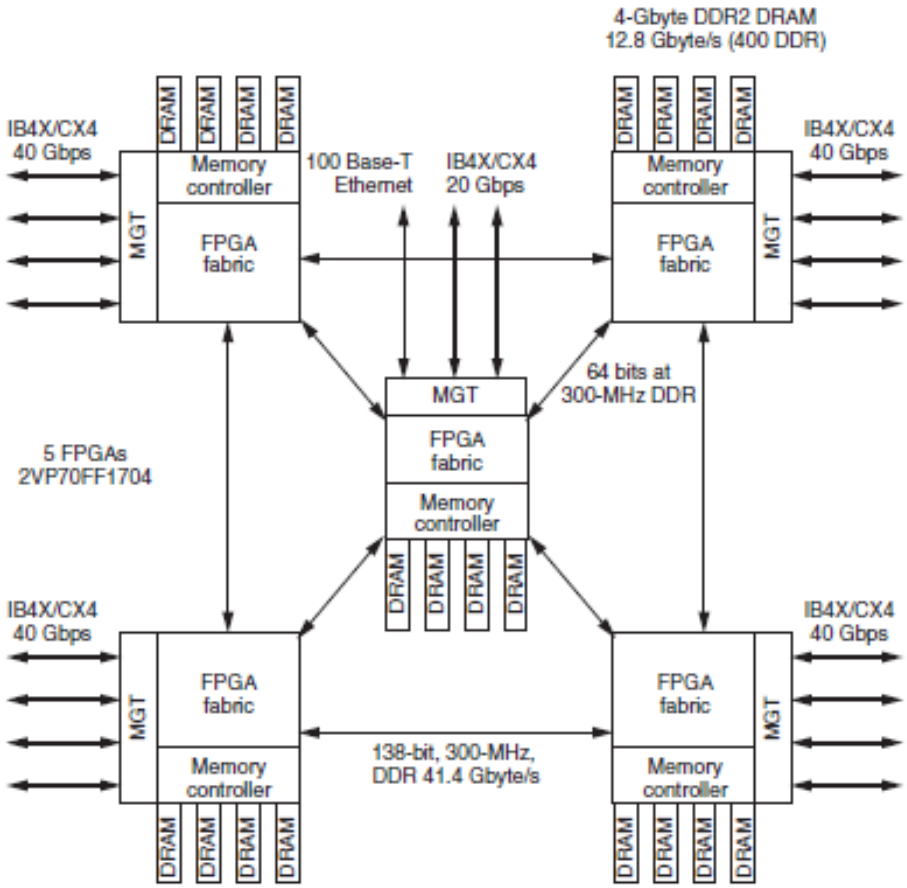


Figure 3.6. Block Diagram of BEE2 Computer Module (C. Chang et al., 2005)

Each computer module has five Xilinx Virtex 2 Pro 70 FPGAs directly connected to DRAM memory modules with a total capacity up to 4 GBytes per FPGA. The central FPGA is programmed as a CPU and performs functions of control module. The rest of the FPGAs are used for computation. According to experimental results, such architecture can outperform DSP chips by a factor of 10 for the Correlator application.

However, the BEE2 is a proprietary CPU+FPGA solution with custom hardware, I/O and memory interfaces. A hybrid system developed on a commodity CPU with off-the-shelf available FPGAs would offer a much more flexible and affordable framework for application development. The following section discusses the system built with COTS components and yet targeting similar computational performance as the BEE2.

3.2 Proposed High-Performance Hybrid DSP System

As a building block for the high-performance hybrid DSP system, COTS personal computer (PC) is one of the most applicable options. Modern PCs feature a wide variety of high-speed communication interfaces for connecting external board with

reconfigurable hardware and contemporary multi-core CPUs yield substantial processing power. Most importantly, PCs are widely spread and easily accessible.

Vendors of reconfigurable hardware issue their products with many interfaces and features:

- On-board memory of various sizes and data exchange rates (DDR SDRAM and DDR2 SDRAM, QDR, SRAM, etc).
- A wide variety of available communications interfaces: SFP, HSMC, PCI family, HyperTransport, Ethernet family etc.
- The hardware is supplied with different range of development tools, IP cores, simulation and debugging software.

For the proposed high-performance hybrid DSP system, the reconfigurable hardware can be plugged in a host PC via supported high-speed interfaces such as PCI Express. Figure 3.7 shows an example of the proposed high-performance hybrid DSP system.

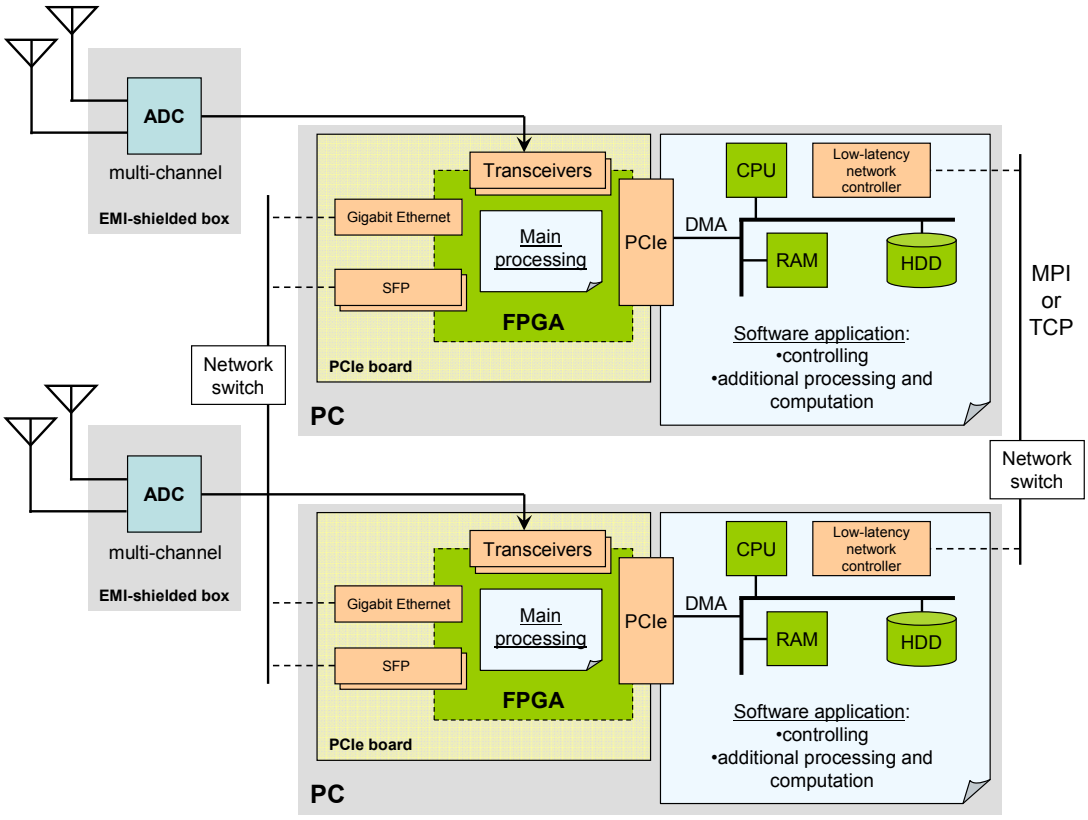


Figure 3.7. Framework for High-Performance Hybrid DSP System

The example system in Figure 3.7 captures data coming from sources (antennas) via two multi-channel ADCs. Further, the data is passed directly to FPGA processing

cores via high-speed on-chip transceivers. The FPGA boards are plugged into the PCIe interfaces of two PC boxes. One of the principal features of the proposed system is the capability to process live data in a real-time fashion.

The diversity of the on-board interfaces in Figure 3.7 demonstrates how these interfaces can be utilised for creating scaled and more efficient systems. If required, communication between two PCIe FPGA boards can be established via either SFP interfaces or Gigabit Ethernet. Similarly, communication can be organized between two PC boxes using separate network controllers (eg by MPI), which allows to carry out complex and distributed high-performance DSP applications. Ultimately, the proposed system is highly scalable: PC and ADCs boxes in Figure 3.7 can be regarded as building blocks for hypothetical DSP systems capable of performing various computationally intensive tasks, eg aperture synthesis for large-scale antenna array. The number of channels, resolution, and bandwidth can be scaled up by using multiple CPU+FPGA boxes.

3.3 Chapter Summary

The above discussion outlines the benefits along with the challenges of the joint usage of commodity processing means (ie CPUs) and reconfigurable hardware (ie FPGAs). Albeit, the effectiveness of deploying FPGAs in high-end signal processing applications also involves a considerable amount of complexity in developing designs in FPGAs as it was described in section 3.1.2. Firstly, and most importantly, because of the fact that hardware description languages (eg VHDL and Verilog HDL) are low-level languages describing hardware behaviour, they are generally challenging to work with. Moreover, unlike conventional software development, the process of hardware design requires careful parallelism consideration, ie in FPGAs all state transitions occur simultaneously according to a specified clock.

The initial intention of a software developer working with FPGAs is to use “standard”, conventional, and typical programming design tools. In other words, a certain degree of abstraction from hardware is desired, which ideally will allow a developer to focus on a functional part of the design rather than the implementation details (Andrews et al., 2004; Fingeroff, Gardner, & Hogan, 2007; Wain et al., 2006). Currently several approaches exist, which allow utilizing conventional methods of programming for hardware design.

The next chapter outlines the methodology applied in his work, which addresses these issues.

CHAPTER 4

Methodology and Design Flow

We can't solve problems by using the same kind of thinking we used when we created them.
—Albert Einstein

This chapter outlines the overall project roadmap towards the proposed high-performance hybrid DSP system. This is followed by a selection of the respective hardware in support of the outlined project design flow. A selected development hardware platform defines the development software required for each stage of the project design flow.

4.1 Project Design Flow and Methodology

In order to evaluate and demonstrate the capabilities of the high-performance hybrid DSP system proposed in 3.2 the following project roadmap was established:

Stage 1. Feasibility study of cross-correlation implementation using a traditional hardware development environment. Create a simple correlator model in HDL and evaluate development effort.

Stage 2. Implement software (in C code) multi-channel cross-correlation of a model signal with added non-coherent noise. Define problem size (correlator lags and number of channels). Measure performance of the software correlator on conventional CPU.

Stage 3. Using integrated support of C-to-HDL tool on XD1000 development system convert the correlation software program into synthesizable RTL design. Then the hardware correlator processes the same simulated signals and its performance is measured.

Stage 4. Using applicable PCI Express FPGA board develop I/O framework for one PC module of the high-performance hybrid DSP system discussed in 3.2.

Stage 5. In order to supply the input data into hardware design and maintain control and management functions, develop a relevant software control application.

To implement the outline stages of this project design flow, appropriate development hardware should be selected. A reasoning of the hardware equipment choice is given in 4.2. That is followed by a discussion on selected development software products in support of the development hardware.

4.2 Development Hardware Platform

The market of reconfigurable hardware is rapidly expanding with new devices coming out regularly, each with more advanced capabilities and larger resources. Presently, there are two main market leaders in FPGA area – Xilinx and Altera. Although, there are other FPGA-chip manufacturers like Lattice Semiconductor, Actel, and Atmel, their products are not as widely supported as Xilinx's and Altera's. Other FPGA manufacturers merely employ the chips created by the aforementioned vendors. The FPGA chips of both vendors drastically differ in terms of available logic elements, available I/O pins, on-chip memory blocks, embedded DSP multipliers, PLLs, etc.

There is a history of using Altera's products in Auckland University of Technology. Existing licence agreements, established development environments, and accessibility at the time of research of Altera's devices, pre-determined the choice of the hardware required to implement project design flow in 4.1

4.2.1 Nios II Development Kit Cyclone II Edition

To estimate traditional development process of the selected application of multi-channel cross-correlation for reconfigurable hardware (Stage 1 in 4.1), an initial feasibility test was undertaken. A trial hardware correlator's design was implemented. A functional block diagram of Nios II development kit is given in Figure 4.1.

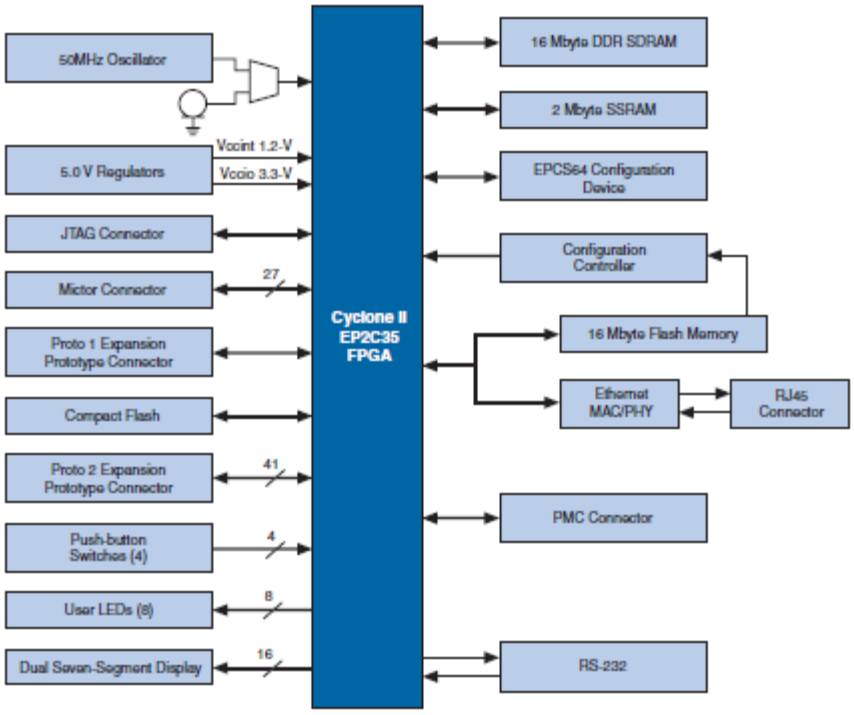


Figure 4.1. Top view of the Nios II development kit (Altera Corporation, 2007c)

The trial hardware correlator's design was implemented on Nios II Development Kit Cyclone II Edition from Altera Corporation. This board has Cyclone II EP2C35F672 FPGA, which is a low-cost device featuring 33,216 Logic Elements (LEs), 483,840 total RAM bits and 35 embedded multipliers. The board also has MAX configuration control logic, 2 MB SRAM, 16 MB DDR SDRAM, 10/100 Ethernet connector, serial RS-232 interface and other features. Particularly, it targets developing system-on-a-programmable-chip (SOPC) designs and supports Altera's Nios II family of embedded processors. This kit is an ideal environment for initial experiences with FPGAs in general and for developing cost-sensitive embedded applications. Most importantly, the whole development environment was already established and accessible at the time of conducting this test. See the full details of this development in 5.2.1.

4.2.2 XD1000 Development System

To implement Stage 3 of the project design flow in 4.1 XtremeData's XD1000™ development system was used. This system employs CPU+FPGA architecture by comprising one COTS AMD Opteron processor and one FPGA-based In-Socket Accelerator™ – Stratix II EP2S180 device plugged into one of the processor sockets of Linux-based PC tower. The FPGA uses available motherboard infrastructure creating a

full-featured CPU+FPGA architecture. Block diagram of XD1000 development system is given in Figure 4.2.

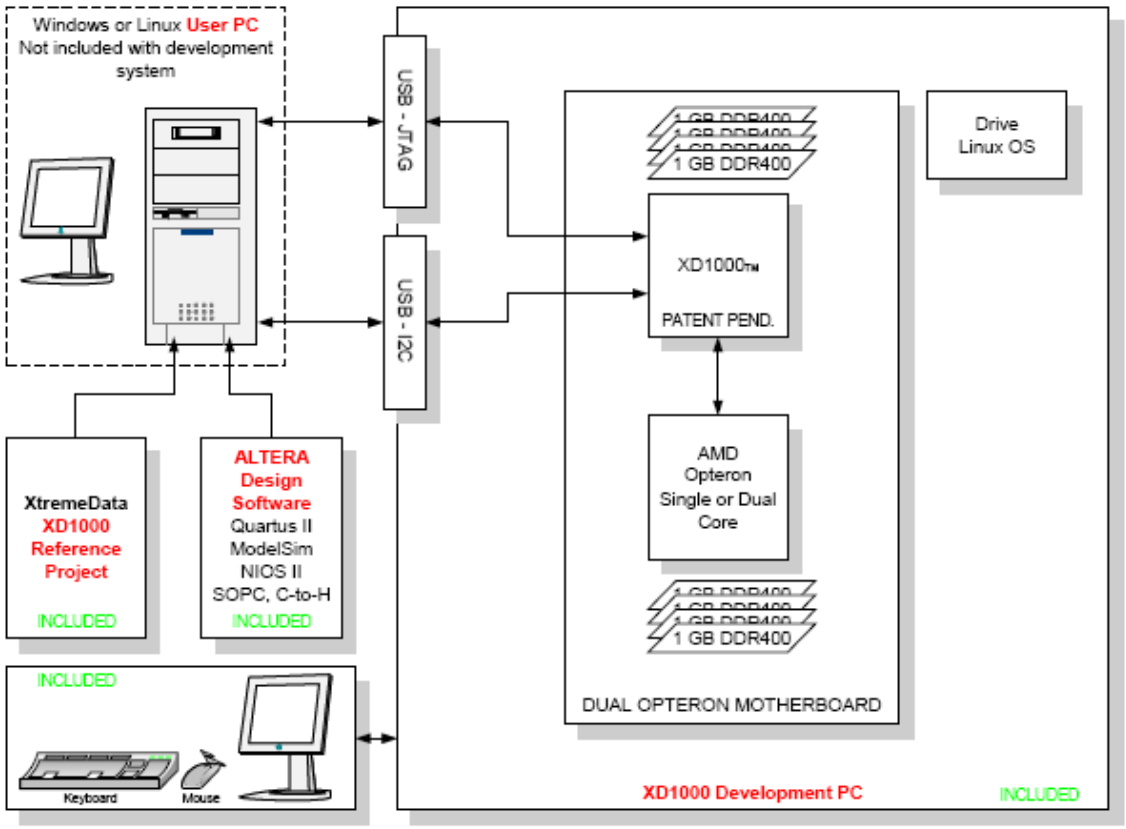


Figure 4.2. Block diagram of XD1000 development system (XtremeData)

Communication between the EP2S180 and Opteron is maintained by two HyperTransport links, 3.2 GB/s each. Such high-speed low-latency interface with a high-bandwidth data flow is necessary for tightly coupled acceleration of an application. The reconfigurable hardware is Stratix II EP2S180F150C3 chip with substantial resources – 179,400 LEs, 9,383,040 of total RAM bits and 384 18×18-bit multipliers. The FPGA can be programmed via a USB cable and up to four configurations can be stored in XD1000 onboard memory. The system also supports a power-up self-configuration scheme common for many FPGA devices. The development system comes with traditional Altera's development tools – Quartus II, SOPC Builder. A reference design is also provided to aid the development efforts.

4.2.3 PCI Express Development Kit Stratix II GX Edition

For implementing Stage 4 of the project design flow in 4.1 PCI Express Development Kit Stratix II GX edition from Altera Corporation (Altera Corporation, 2007d) was considered the most applicable tool at the time when the choice was made (September –

November 2006). Figure 4.3 shows a functional block diagram of the Stratix II GX PCI Express development board and introduces most of the board's interfaces and features.

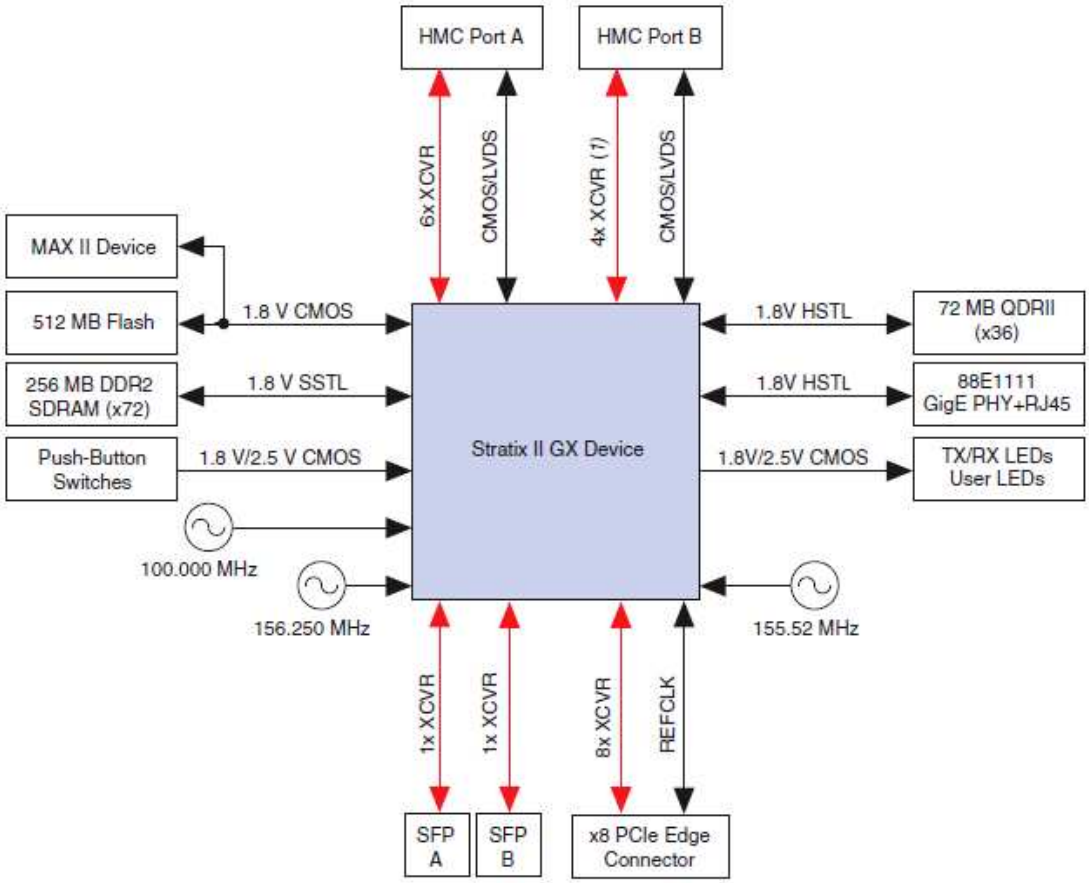


Figure 4.3. Stratix II GX PCI Express Development Board (Altera Corporation, 2007d)

This board possesses three crucial features for the targeted high-performance DSP applications: the ability for data acquisition (eg through High-Speed Mezzanine Connectors (HSMC or HMC) with six on-chip transceivers routed to them – J1 and J2 in Figure 4.4), PCI Express or PCIe interface for data exchange and is reasonably low cost compared to the number of available interfaces. The board has PCIe ×8 interface, which allows it to be plugged into the PCIe bus of a commodity PC and achieve a data exchange rate of up to 250 MB/s in each lane in each direction or up to 2 GB/s in each direction in total for the board. The PCIe interface surpasses the majority of the communication interfaces mentioned above: thus, for example PCI-X 1.0 interface achieves only 1,066 MB/s at 133 MHz (PCI Special Interest Group, 1999) or 1 GB per second for the Gigabit Ethernet. Top view of the development board is given in Figure 4.4.

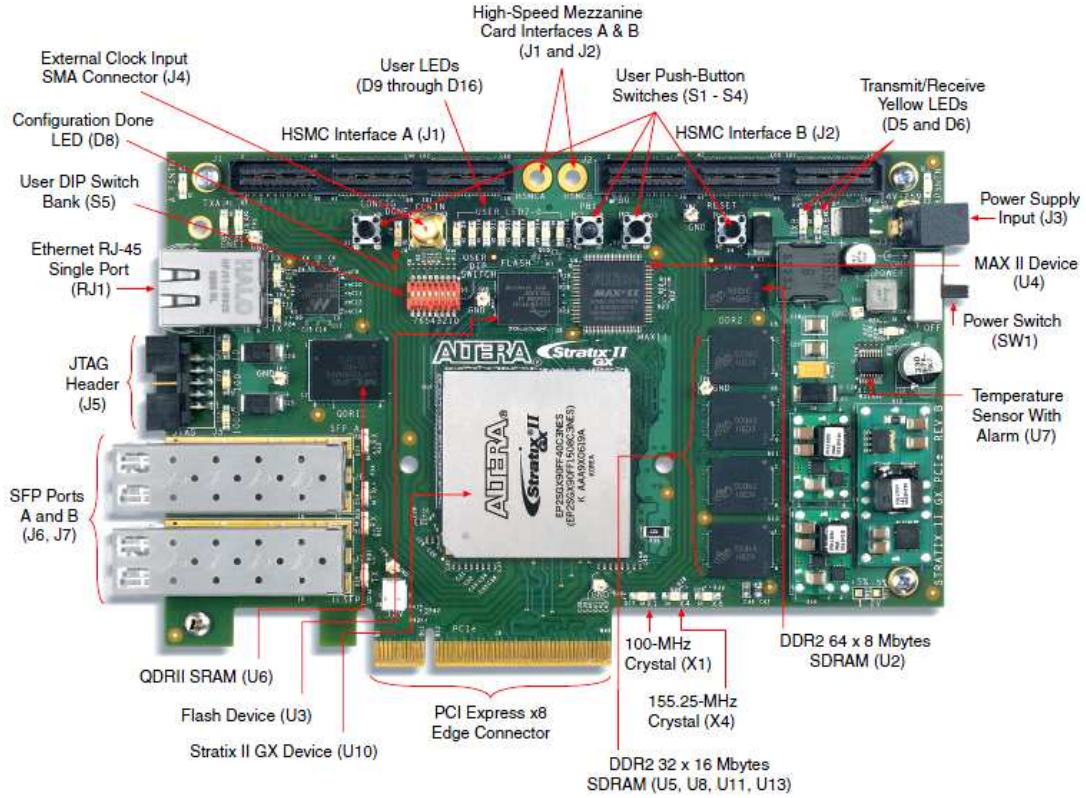


Figure 4.4. Top View of the Stratix II GX PCIe Development Board (Altera Corporation, 2007d)

The featured FPGA - Stratix II GX chip (U10 in Figure 4.4) is a very powerful chip and the Stratix II family was considered a flagship of Altera's devices at the time of development. The chip used on the board has 90,960 LEs, 16 transceivers, more than 4 Mb of on-chip RAM and 48 DSP blocks. The latter ones can be configured into a dedicated circuitry which can perform multiplication, multiply-accumulate (MAC) and multiply-add functions with high efficiency. Such a feature is particularly beneficial for DSP applications in general and for the proposed system in particular.

Another advantageous feature of this board is the high-speed Mezzanine connectors, which are routed to the transceivers inside the Stratix II GX chip. This option allows direct, intermediate data acquisition (eg from ADCs) and further streaming of it into the chip for immediate processing. Hence, such workflow implies the possibility of employing the FPGA chip in real-time processing. This aspect significantly increases the capabilities range of this board in the DSP applications domain. Although, real-time correlation is not targeted for implementation in this particular project, it remains as one of the objectives for the future work (see 7.2).

In this work the PCI Express Development Kit Stratix II GX edition is used to develop an I/O framework for the targeted high-performance hybrid DSP system. As a jump-start and to ease the development efforts, the PCI Express to DDR2 SDRAM

Reference Design from Altera Corporation will be used as a foundation for the framework (see 5.4.1). The developed framework on the PCI Express Development Kit also serves as a building block for a scalable projected system for performing high-performance DSP applications (see 3.2).

Stage 2 and 5 require traditional software development tools and have been implemented on a conventional PC.

4.3 Development Software Tools

Once the hardware platforms for implementing the stages of the project design flow in 4.1 were selected, respective software development tools had to be selected. Stage 1 and Stage 4 require traditional hardware development tools, which are discussed in 4.3.1. Whereas Stage 3 employs new hardware design methodology, which is discussed in 4.3.2. Section 4.3.3 discusses software development tools required for Stage 2 and Stage 5.

4.3.1 FPGA Development tools

Presently, there are a number of tools which are capable to work with FPGAs. Firstly, the development tools which are supplied by two main FPGAs' vendors Xilinx and Altera – ISE and Quartus II. For the selected Altera's PCI Express Development Kit Stratix II GX edition and XD1000 development system, the major development tool is Quartus II software.

At the very early stage of the project development, Altium Designer (Altium Limited, 2008) was considered as the primary instrument in design development. However, at the inception of the project the most recent version 6.1 (early 2007) of Altium Designer demonstrated a significant disadvantage for this project: there was no support for Altera's IP cores (MegaCore functions or Megafunctions in Altera terms) through which the majority of the interfaces are implemented. Besides, Altium did not offer any link to the MATLAB environment unlike Quartus II, which has the DSP Builder. Therefore, a decision was taken to revert to Quartus II software.

Quartus II development software is a complete design environment for developing hardware designs for Altera's hardware products. It supports complete design flow:

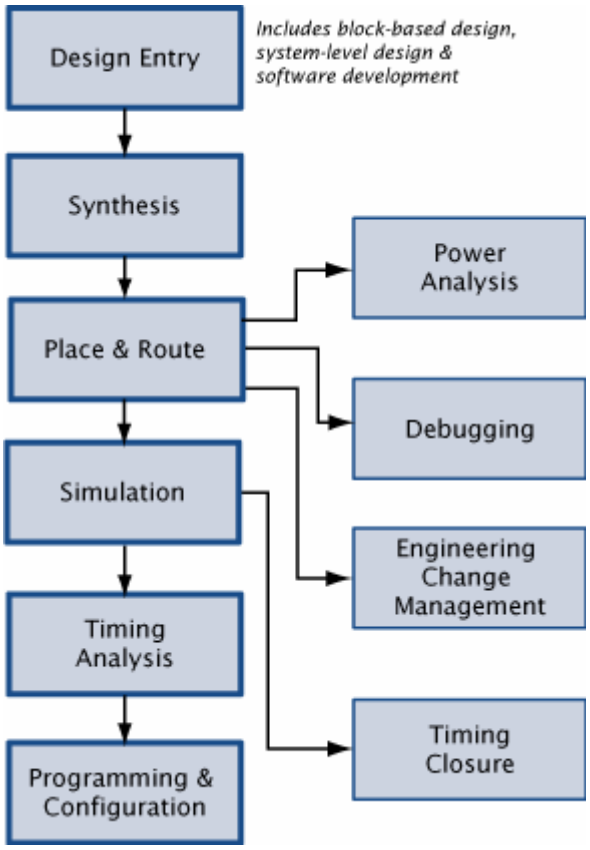


Figure 4.5. Design Flow in Quartus II Software (Altera Corporation, 2007e)

Besides, Quartus II offers various design entry methods: from low-level hardware description languages (Verilog, VHDL or AHDL - Altera Hardware Description Language, propriety HDL language of Altera), to high-level visual means (schematics and block diagrams). It comprises several unique design-aid features, eg Incremental Compilation aimed for reducing compilations or SignalTap Logic Analyzer for on-chip design debugging purposes. The latter utility captures internal data and service signals based on preset triggers and stores them for the following analysis (Altera Corporation, 2007a). It was extensively used for debugging I/O framework design (5.4.2). The majority of the work was done in Quartus II version 7.2 with compilations for the XD1000 development system done in Quartus II version 8.0.

Apart from Quartus II, several other of Altera's development products were also employed: for Stage 1 of the project design flow in 4.1 the DSP Builder was used to test the concept of correlator implementation in FPGA. This software operates as the Translator in Figure 4.7. This is achieved by amalgamating the Simulink/MATLAB environment with Quartus II projects using a specific DSP Builder Advanced Blockset – a Simulink library developed by Altera. The precise role of the DSP Builder is revealed in 5.2.1.

Quartus II development set-up follows Altera's workflow recommendations and is shown in Figure 4.6.

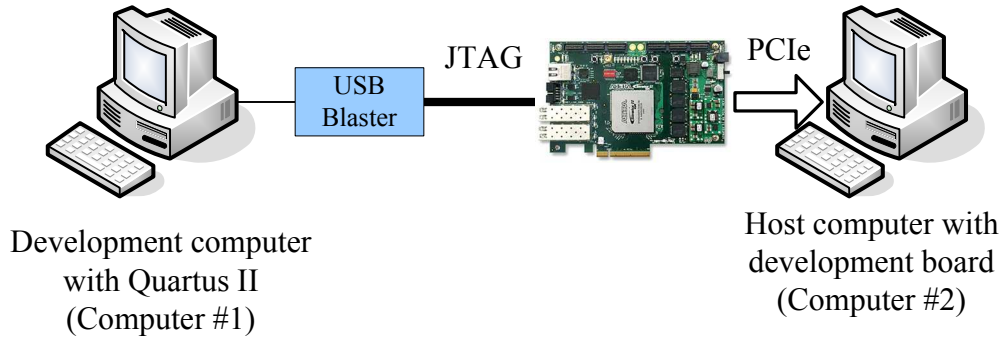


Figure 4.6. Development Setup for PCI Express Development Kit Stratix II GX edition

PCI Express Development Kit is plugged into a host computer (Computer #2) via the PCIe connector. The Stratix II GX FPGA is configured by means of JTAG. The board also features MAX II CPLD (U4 in Figure 4.4) which is also used for FPGA configuration from the pre-loaded on-board flash memory (U3 in Figure 4.4). The development computer (Computer #1) carries development software and programs the board. After the FPGA is programmed, Computer #2 requires a reboot to instantiate the device in the operating system. It hosts software control applications developed with WinDriver applications. In the case of the Nios II Development Kit, the development computer acts as the host computer as well and the kit is connected via the USB blaster.

The ModelSim Altera edition from Mentor Graphics was also used during one of the development stages in the project. This tool offers a comprehensive, functional and behavioural simulation and debug environment for complex FPGA designs. Particularly, for this project this tool will be used for testing and debugging of certain IP cores from Altera. This is also discussed more broadly in Chapter 5.

4.3.2 New Hardware Design Methodology

To implement Stage 3 new design methodologies different from the traditional, has to be applied. Traditional top-down design methodology can be divided into two separate design domains: algorithm development and system implementation. The nature of both domains is entirely different even to the point of contradiction. Algorithm or system developers work in high-level programming environments such as MATLAB and Simulink and rarely C-based languages. The primary goal of an algorithm developer is algorithm accuracy and system functionality. A system development team outputs

system descriptions to a hardware development team. In turn, hardware design teams implement the specifications created by the systems engineers and algorithm developers in the targeted hardware: whether it is an FPGA, ASIC, SOC, DSP, microprocessor, etc.

Throughout all of the hardware design stages various design verification routines (checks, simulations and analyses) are performed (Meyer-Baese, 2004b). These verification routines are a part of an iterative, communication process between algorithm developers and hardware engineers with the purpose of refining the algorithms and system architecture until all the design requirements are met. Quite often, this process takes many numbers of man-hours to track down a relatively simple problem in the design, mainly due to inefficient interaction between the two domains.

This gap has been acknowledged and explored by many researchers (Andrews et al., 2004; Ganousis, 2004; Hill, 2006; Leow, Ng, & Wong, 2006; Meyer-Baese, Vera, Meyer-Baese, Pattichis, & Perry, 2006; Tahernia, 2005; Urbanek & May, 2004). Even two leading FPGA vendors, Xilinx and Altera, acknowledged the missing link between the two development fields and offered their own approaches in this direction: (Turney, Dick, Parlour, & Hwang, 2000) and (Altera Corporation, 2002) respectively. It is envisioned by the majority of the researchers that this can be achieved by establishing a certain medium between the two domains, which will effectively analyze system requirements, automatically create RTL models and so that the rest of the hardware implementation cycle can be performed. This flow is illustrated in Figure 4.7.

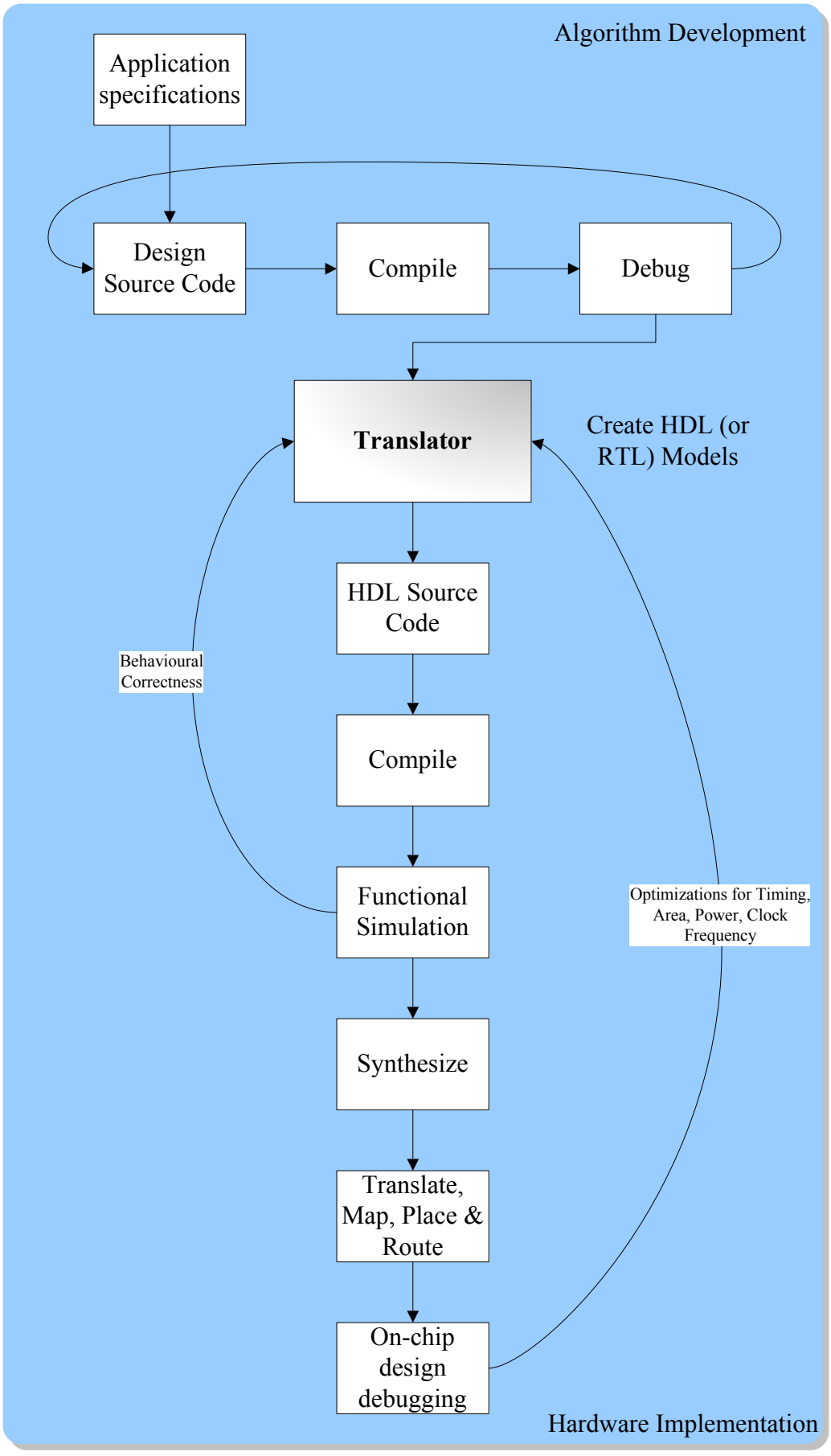


Figure 4.7. New Top-Down Design Flow with Integrated System Level

The crucial part of the new design flow is the "Translator" block. The functionality of this block can be performed by a variety of tools (see 3.1.3). In this work, Impulse CoDeveloper version 3.20.a.5 from Impulse Accelerated Technologies will be used.

Impulse CoDeveloper allows application developing and debugging using C standard development environments, which are then compiled to create outputs (VHDL, Verilog or SOPC libraries) fully compatible with Altera's Quartus II and SOPC Builder. The tool can produce synthesisable HDL-code (VHDL or Verilog) from an input C-code.

As mentioned in 3.1.3 before programming, FPGA is unaware of its input and output capabilities. While porting of software applications into hardware domain, inputs and outputs of the transferred application have to be explicitly conveyed to. and/or expressed in, the selected transferring tool. In the case with Impulse C, a stream-oriented programming model for data movement, processing, and synchronization is used. Conceptually stream-oriented programming is similar to conventional, dataflow programming. However, unlike dataflow, stream-oriented programming offers easier process synchronization by means of buffering and message-passing such non-dataflow concepts as shared memories (Impulse Accelerated Technologies, 2008d). In terms of the programming model of Impulse C, streams communicate with processes, which represent hardware implementation of the converted software application. Each software application can consist of a number of processes, which synchronously and concurrently operate with each other and/or with the external world as defined in the original software operation. The idea of Impulse C programming model is illustrated in Figure 4.8:

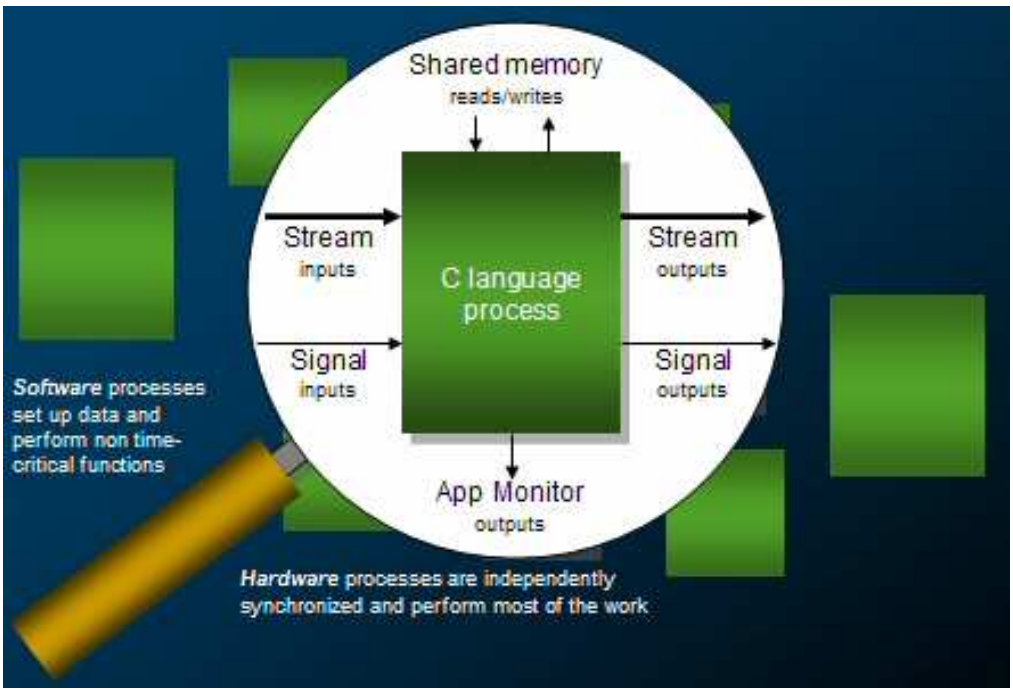


Figure 4.8. Impulse C programming model

Software and hardware processes shown in Figure 4.8 represent complete application. Software process(es) is(are) used only for desktop simulation and is(are) generally employed with inputting/outputing data to/from hardware processing and/or is(are) occupied with non-computationally intensive tasks. The hardware process(es) is(are) translated into HDL descriptions and also participate in desktop simulations. The source file of a hardware process also contains a *configuration function* (co-initialize and co_-architecture_created), which assembles the whole applications, interconnects processes, specifies shared memory locations, etc. Software and hardware processes are stored in software (*_sw.c) and hardware (*_hw.c) source files or modules respectively. For supported systems with respective Platform Support Package, the software module can also interact with the synthesized and programmed hardware module. For example, in the XD1000 development system the software program is copied on the target XD1000 server and can start working with the hardware or can be dynamically modified as required.

Impulse CoDeveloper is based on Impulse C – a proprietary subset of C programming language with a compatible function library, which allows compiling directly into optimized logic ready for synthesis and programming of popular FPGAs. However, CoDeveloper does not offer a “push button” solution for generating ready-to-be-programmed hardware designs from complete C-projects. Rather the application’s

code has to be comprehensively adapted and re-worked to successfully map into reconfigurable hardware and efficiently extract algorithm's parallelism.

Moreover, to increase application speed-up in hardware Impulse C supports additional level of control over the generated hardware code via predefined *pragmas*.. The first pragma is `#pragma CO PIPELINE`. For loops invoked with this pragma, CoDeveloper attempts to parallelize statements within the loop trying to reduce the number of clock cycles required to process the entire pipeline (Impulse Accelerated Technologies, 2008e). Another pragma – `#pragma CO UNROLL` unrolls a loop. Unrolling a loop implies that the code within the loop is duplicated in hardware as many times as there are iterations in the loop (Impulse Accelerated Technologies, 2008a). Unroll pragma can be applied to loops where the number of iterations is known during the compile time. Theoretically, it might significantly reduce the execution time if the number of loop iterations is relatively small. Otherwise, applying this pragma might increase logic utilisation drastically.

As mentioned in 3.1.3, high-level programming languages like C were never designed to configure hardware logic. Contemporary high-level tools for FPGA programming tackle this problem in different ways. Impulse C treats the input C-code in the form of *blocks*. The blocks can be divided by either:

- loop body,
- a chain of control statements (co_stream_open, co_stream_close, co_stream_read, etc.),
- switch,
- conditional statements.

For each determined block of C code, the Impulse C's optimizer will estimate the minimum number of instruction stages, ie groups of C statements which can be executed in parallel. As long as each block can consist of multiple stages,several clock cycles might be required to execute the given block.

Impulse CoDeveloper features several simulation tools for verification and analysis of the generated HDL code. Stage Master Explorer tool illustrates how every block and stage of software application was realised in hardware. Stage Master Debugger can perform sequential execution of the generated application on a cycle-by-cycle basis.

Although Impulse provides an estimation of the required logic resources in terms of adders, multipliers, comparators, DSP blocks, etc, the precise resource utilisation is determined only by the vendor's compilation tool. Similarly, the maximum clock rate in hardware is determined by the FPGA's synthesis tool, where a sequence of optimisations, reductions, and combinations are applied to the compiled logic.

What is more important within the confinements of the given work is support of XD1000 by CoDeveloper. The XD1000 PSP for Impulse CoDeveloper extends the capabilities of XD1000: it provides an automated process of generating software and hardware modules that execute on AMD Opteron and XD1000 co-processor modules respectively. The benefit of this integration enables creation of high-speed, accelerated designs working across software and hardware domains from a standard ANSI C development environment. Once a design is tested and verified in Impulse CoDeveloper environment it is exported in a highly-automated process to a complete Quartus II project ready for synthesis (according to the flow in Figure 4.7). Development set-up in this case is the same as for the PCI Express Development Kit in Figure 4.6. The host computer is an XD1000 system and connects to the development computer via an USB blaster.

There are, however, certain limitations and reservations in the current XD1000 PSP version (Impulse Accelerated Technologies, 2008b):

- DDR SDRAM available on XD1000 module is not supported.
- The HT-core is limited to 8-bit (instead of available 16) and uses approximately 400 MB/sec full duplex (800 MB/sec aggregate) versus an available 3.2 GB/s per each link.
- One of the most serious limitations is that software-hardware communication via streams is not Direct Memory Access (DMA) and polls CPU for each request, which significantly reduces performance and yields only 2 MB/sec of bandwidth in total.
- All user logic in hardware is constrained to 100 MHz.
- Only one concurrent software process on the target is supported.
- The maximum stream data width supported is 32 bits.
- The maximum shared memory data width supported is 64 bits.

Impulse Accelerated Technologies suggest alleviating the limited bandwidth of streaming interfaces by employing supported shared memory communication. This communication is much faster than a streaming approach and yields up to 800 MB/sec. The precise effect of the above limitations is covered in 7.1 Discussions. The targeted multi-channel correlation application will be developed with CoDeveloper, exported and compiled in Quartus II and executed on XD1000.

4.3.3 Software Development Tools

Successful work with the PCIe FPGA board in Windows operating system environment is supplied by a software control application developed with the Jungo WinDriver PCI/PCI Express/PCMCIA development tool (Jungo Ltd., 2008). This product features a simple process of creating a hardware driver for any device working via one of the supported interfaces (PCI, PCI Express, PCMCIA, etc.). Using a GUI interface WinDriver automatically detects hardware resources of the plugged device and generates a respective driver code skeleton for a specified development platform (MS Developer Studio, Borland C++ Builder, etc). The sample application can be modified further to suit the specified requirements. WinDriver also provides many generated example applications. One such application is a diagnostics application for accessing Altera Stratix II GX PCI Express Development Board – `altera_diag`. This simple application provides read and write operations to Altera memory and I/O registers. For the given project, this application can be efficiently and easily adapted to provide necessary input and output communication with the Stratix II GX PCI Express Development Board via PCI Express interface. The code of the `altera_diag` application was modified in Microsoft Visual Studio 2005.

The sampled model signal was created and recorded using MATLAB. The respective MATLAB script generating the signals is given in Appendix A2. As was already mentioned, Simulink was also used together with a DSP Builder utility for early feasibility estimation of correlation in reconfigurable hardware.

The reference program of the 32-lag cross-correlation C-code (5.3.1) was developed in the Microsoft Visual Studio 2005.

4.4 Chapter Summary

This chapter introduced project design flow, outputs of which are targeted to estimate the capabilities of the proposed hybrid DSP architecture (3.2). The project roadmap is composed of five stages:

Stage 1. Implement trial cross-correlator design with traditional hardware development tools.

Stage 2. Develop the reference software multi-channel correlation program.

Stage 3. Convert the reference program using Impulse CoDeveloper into accelerated hardware design to execute it on XD1000 development system.

Stage 4. Develop I/O framework on one PC module of the proposed high-performance hybrid DSP system (3.2).

Stage 5. Develop software control application with control and data management functions for I/O framework.

These respective stages define the development hardware and the respective software applied in each stage:

Stage 1. Nios II Development Kit Cyclone II Edition with standard Altera development tools.

Stage 2. COTS development PC with Microsoft Visual Studio. MATLAB software to generate model signals.

Stage 3. XD1000 development system with integrated Impulse CoDeveloper support.

Stage 4. PCI Express Development Kit Stratix II GX edition with standard Altera development tools.

Stage 5. COTS development PC with Jungo WinDriver PCI/PCI Express/PCMCIA development software and Microsoft Visual Studio.

CHAPTER 5

Implementation

The only place where success comes before work is in the dictionary.
—Donald Kendall

This chapter presents the implementation process of this work. First, the chapter outlines the overall implementation flow of the project. The following section defines the approaches of this work: the first initial stage, estimation design of hardware two-channel correlator and positioning of correlation problems are presented. The remainder of the chapter describes the rest of the implementation stages, which form two major parts: correlator implementation part and I/O framework part.

5.1 Implementation Flow

The project's design flow discussed in section 4.1 involves C-to-HDL tool – Impulse CoDeveloper. Therefore, the project's implementation employs new top-down design flow considered in 4.3.2.

Figure 5.1 demonstrates the implementation flow for the given project, highlighting the respective delivered outputs of the project (blocks coloured in grey). These outcomes are more broadly covered in Chapter 6 and discussed in Chapter 7.

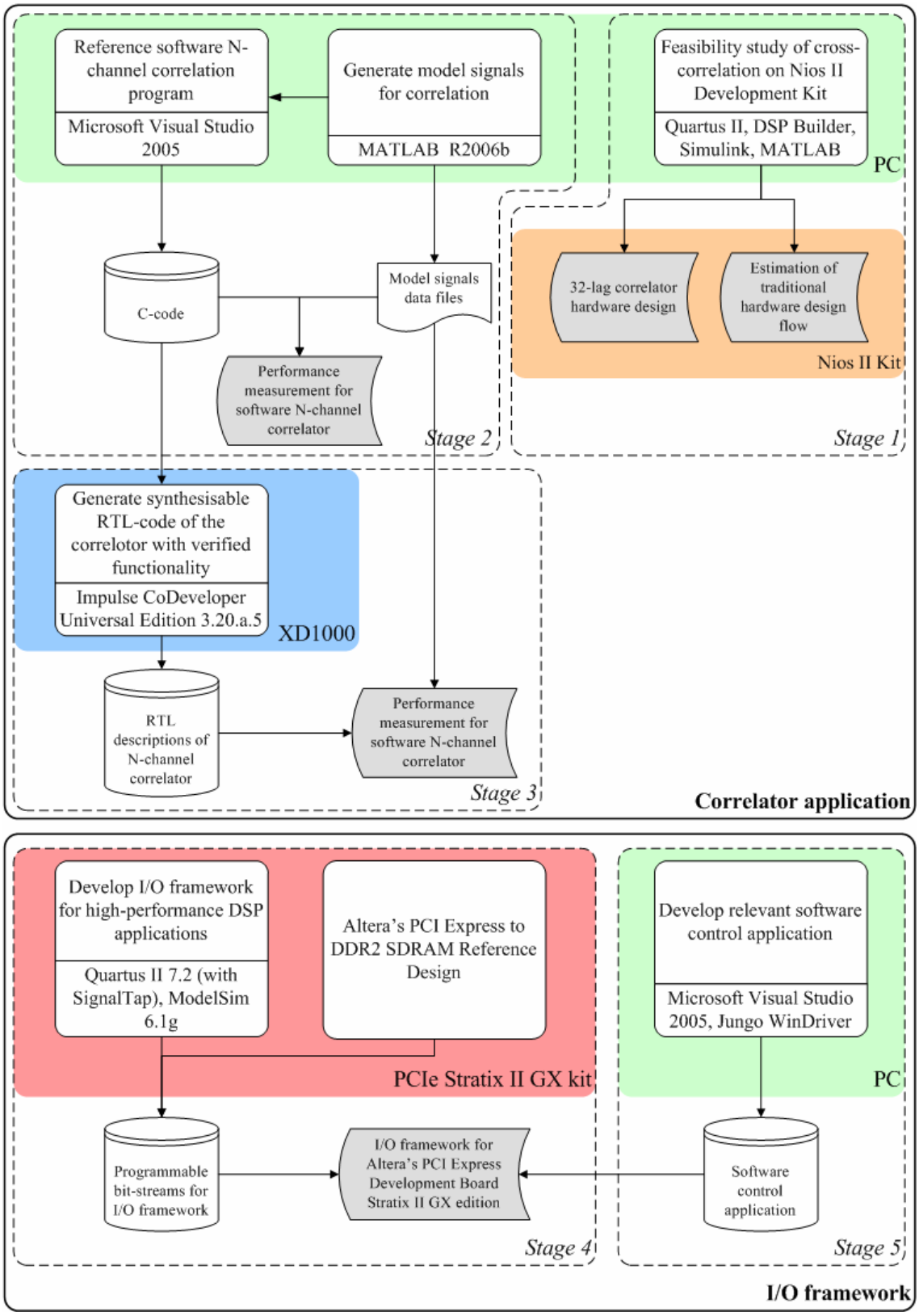


Figure 5.1. Implementation Flow

The flow demonstrated in Figure 5.1 also highlights all of the software products used at particular stages of the design. Broader introduction of the software tools is given in 4.3.

A common problem for any design employing an FPGA(s) for a DSP application is the development of the necessary I/O interfaces. As discussed in 3.1.2, FPGAs have no knowledge about their respective input and output interfaces, which is one of the major problems in FPGA designs (Wain et al., 2006) and often requires as much attention from a development team as the core problem itself (Romein et al., 2006a). This project is also no exception. The workflow in Figure 5.1 delivers outcomes in two domains:

1. Correlator design. First, using C programming language in Microsoft Visual Studio and then transferring the very same design into RTL by means of Impulse CoDeveloper considering necessary alterations.
2. I/O framework. Developing of the I/O interfaces necessary for DSP applications. The I/O framework for this project will use Altera's PCI Express to DDR2 SDRAM Reference Design.

Implementation details of correlator and I/O framework are described in sections 5.3 and 5.4 respectively.

5.2 Defining Approaches

Before actual implementation, a trial correlator design in HDL was undertaken. This design employed the existing simulation chain Quartus II – DSP Builder – MATLAB. This design was used as a departure point for the actual correlator design of Stage 2. It was also used as an estimation of the efforts and time required to develop a high-speed multi-channel cross-correlator on an FPGA using traditional design methodology with conventional hardware developing tools (HDL coding, schematics, etc).

5.2.1 Trial Hardware Correlator's Design (Stage 1)

The trial correlator design is based on Altera's parameterized multiply-accumulate megafunction – altmult_accum. This MegaCore function is used as a foundation for the correlator's lags. This function consists of a single multiplier feeding an accumulator. The whole correlator design was tested and verified in a DSP Builder environment – an autocorrelation function of 5 kHz wave was computed. The code of the Correlator's top-level entity and the schematic of correlator lag are given in Appendix A1. An example output of this simple correlator is given in Figure 5.2:

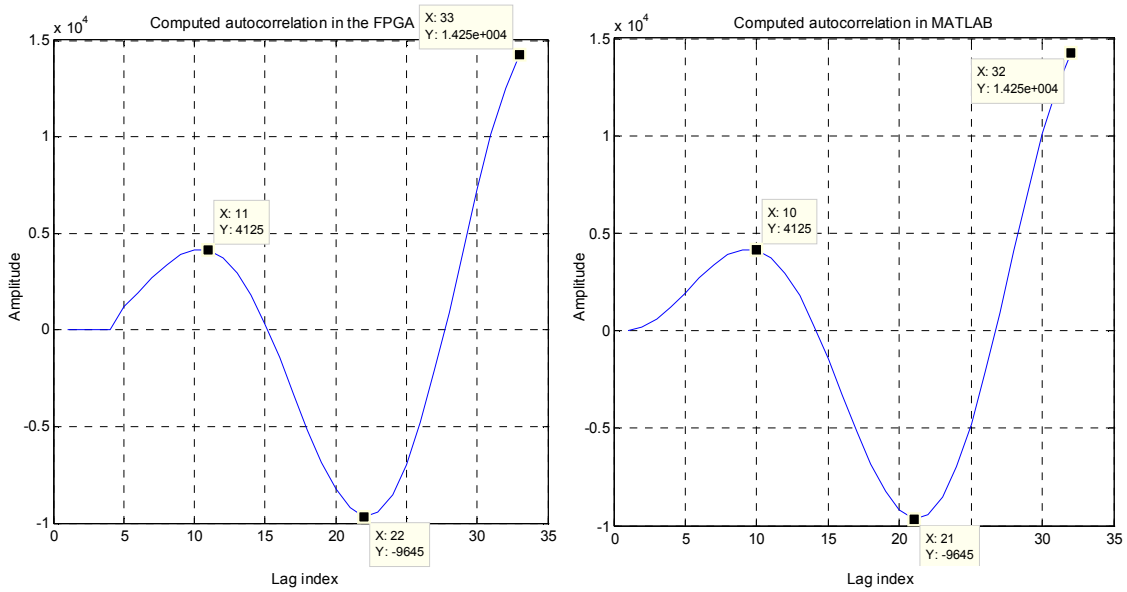


Figure 5.2. Autocorrelation Function of 5 kHz Sine Wave – computed in the FPGA (left) and computed in MATLAB (right)

The 32-lag hardware correlator performed autocorrelation of 5 kHz sine signals generated in a Simulink environment and fed directly into the hardware. Amplitude is one unit and sampling rate is 20 kHz. The correlator's output was captured as a MATLAB variable and plotted (left) against a computer-simulated correlation function (right) using `plot` function. For convenience, values at the peak points are presented on data tabs.

The plot demonstrates only half of the output function since the autocorrelation function is even. For the reason that this design was not intended to be a complete application, some design flaws exist. For example, the current design ensures that the output latch goes high only for one clock cycle – `LastLatch`, `LastLastLatch` and `SynchronousLatch` (see Appendix A1). This might be redundant and and might exaplain discrepancies in the initial values of the autocorrelation functions. The Simulink model used for testing the developed correlator design is presented in Appendix A2.

An attempt was made to replicate this design on a Stratix II GX board. This design revealed a critical deficiency in simulation capabilities of the Stratix II GX PCI Express Development Kit: the board features MAX II CPLD, which can be used for power-up configuration of the main Stratix II GX chip. Thus, the two devices share the same JTAG chain – see Figure 5.3.

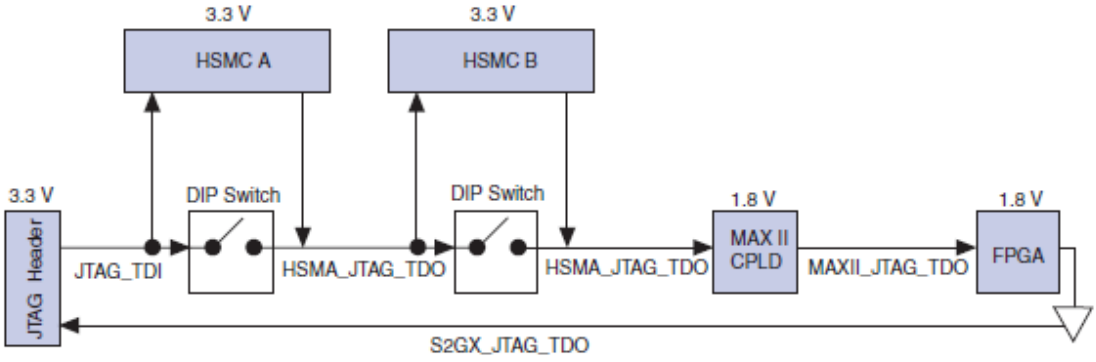


Figure 5.3. JTAG Chain Connections in Stratix II GX PCI Express Development Board (Altera Corporation, 2007d)

This disturbs the DSP Builder's simulation data transfers through this chain. Therefore, the whole DSP Builder simulation environment is inaccessible to any Stratix II GX PCI Express Development Board, which might hamper some DSP developments on these devices.

This correlator design is not intended as a final application in this project. Rather, it is a reference point and trial attempt to help define design approaches and estimate design efforts.

5.2.2 Problem Positioning for Stages 2 – 5

The cross-correlation considered in this work adheres to Equation (2.1). Hence, the application implemented within the confinements of this project does not fall in the exact definition of cross-correlation in terms of (for example) radio astronomy, where mathematical computation of Equation (2.1) has to be coupled by FFT either before or after defining the FX or XF correlation respectively as described in section 2.2.2. Nevertheless, this project involves development of an actual digital correlator, therefore, certain requirements and conventions have to be accepted before actual implementation. The following paragraphs disclose parameters and requirements, which apply to the correlator developed in this work.

The core processing application follows the mathematical definition of correlation defined by Equation (2.1). In most applications the number of lags is even and to a power of two. As long as the eventual goal of this work is not to develop a finalised correlator for a given DSP application, but rather to investigate the implementation of a classical DSP problem in a hybrid CPU+FPGA environment, there was no particular reason to create a correlator with a significant number of lags. Besides, the trial design

(5.2.1) revealed that with manually created correlator logic, the Stratix II GX FPGA is able to accommodate up to 1,024 lags in total, which is rather substantial. Therefore, a correlator with 32 lags was considered a sufficient case for the targeted goals in this work.

Moreover, the correlator has to support multi-channel correlation, ie to perform correlation between each pair of the input channels (or signals). For correlation of multiple signals only unique pairs of signals will be correlated (see 2.2.1 for details). Hence, for example, correlation of signal $x[k]$ with signal $y[k]$ will be computed and correlation of signal $y[k]$ with signal $x[k]$ will be not computed since it can be obtained by simple reversing of the former resultant correlation function. In the given work a correlator with six input channels will be implemented, which will produce 15 output cross-products.

In real-life applications (e. radioastronomical applications), the correlator's output is aggregated: while processing the output is accumulated and read after a specified number of input (processed) samples. For example, a two-channel correlator processes digitised, input sequences and outputs results after 10,000 input samples were processed. After output is read, the aggregation of results starts over. By convention, the number of aggregation input samples in this work was accepted to equal 4,194,304. This number was considered practical and convenient for measuring performance results for both software and hardware correlator implementations.

As for the input signals, the following assumptions were undertaken to adhere as closely as possible to realistic correlation requirements. A typical bit width of the input data is 6 to 8 bits: eg ADCs' outputs are typically 6 bit width (Maxim Integrated Products, 2001). Hence, the correlator's inputs width was decided to be 8 bits, which is also convenient for data manipulation in 32-bit Windows OS environment. As stated previously, the correlator's output has to accommodate aggregated results. Therefore, the width of the output of the correlator's cross products was selected to be 32 bits, which is sufficient enough to accumulate 4,194,304 8-bit input samples.

Each correlator’s channel has been with a 32 MHz sinusoidal wave with added white Gaussian noise. The signal-to-noise ratio (SNR) parameter of the MATLAB `awgn` function is set to three to introduce a realistic noise for signals. The input sine wave signals are digitised in 6-bit samples at 256 MHz sampling rate, which also defines the operational bandwidth of the correlator – 128 MHz bandwidth common for many DSP applications involving correlation. The samples have 6-bit width to simulate real-life

data coming from ADCs. These 6-bit samples are padded two bits and fed to 8-bit correlator inputs. MATLAB was used to generate input signals and store them in form of the text files (see Appendix A3 for MATLAB script used to generate the signals). The values of sine waves are rounded to the nearest integer.

The following table summarizes established correlator's parameters:

Table 5.1. Correlator Parameters Summary

Input signals	32 MHz sinusoidal waves with added white Gaussian noise (SNR = 3)
Bandwidth	128 MHz
Input bit width	8 bits
Output bit width	32 bits
Number of lags	32
Read after (number of samples to aggregate)	4,194,304

The following versions of multi-channel correlators will be implemented: 6-, 8-, 10-, 12-, and 16-channel. The parameters and restrictions discussed above apply to the correlator design throughout this work in both, hardware and software domains and for all correlator versions. The next section discloses the implementation details in both domains and elaborates on the flow described in 4.1.

5.3 Implementations for Stages 2 – 5

According to Figure 5.1, both software and hardware implementations will require model signals for processing. Parameters of the model signals are given in Table 5.1.

5.3.1 Reference Software N-Channel Correlation Program (Stage 2)

According to 4.1 the initial task in correlator implementation is to develop a software program, which will perform a multi-channel correlation of model signals. There are many example codes performing correlation, which are written in various programming languages. Since the selected Impulse CoDeveloper tool supports a subset of C, the code was developed in plain C language. The code was developed in Microsoft Visual Studio 2005. The correlation algorithm implemented in this program is straightforward

and strictly follows Equation (2.1). Hence, the code reads all samples of the simulated model signals (stored as text files) and performs correlation of the buffered values:

```
for (lag = 0; lag < Nlag; lag++)
{
        for(i=length/2; i < length; i++)
        {
              result[lag] += x[i]*y[i-lag];
        }
}
```

Figure 5.4. Two-Channel Correlation in C

The code given in Figure 5.4 computes two-channel cross-correlation of signals `x[i]` and `y[i]` and stores the correlation function in `result[lag]`. Based on this two-channel computational core a scalable version of the reference program was developed. Each two-channel core produces one output cross-product of 32 calculated lag values. Depending on the number of input channels, the correlator will produce a certain number of unique cross-products (see Equation (2.2)). The code of the reference program computes cross-correlation for a specified number of channels and a specified number of aggregated samples (see 5.2.2) and is listed in Appendix A4.

The execution time will be measured for different versions of multi-channel correlation computation: 6-, 8-, 10-, 12-, and 16-channels and results are presented in 6.1. Chapter 6. The time is measured using High Resolution Timer (to nanoseconds). The code was developed by L. F. Johson at the Systems Design Engineering University of Waterloo and is freely available from the Internet. It relies on Windows QueryPerformanceCounter() function and is already included in the reference program code – see Appendix A4.

It has to be mentioned that the code of the reference program was not developed with the intention of performing cross-correlation with maximum efficiency for real-life applications. It was developed rather to estimate the rationale of C-to-HDL tools on a basic and common DSP algorithm.

The code of the reference program was used as a starting point for Impulse CoDeveloper design. However, due to specific limitations inherited from FPGA hardware synthesis, certain conventions had to be preserved. For example: there is no recursion for FPGA hardware processes and limited support of function calls, pointers must be resolvable at compile time to static references to specific memory locations and others (Impulse Accelerated Technologies, 2008c). Although these constraints did not hamper development of the software correlator significantly, they were taken into account during conversion of the developed software application to synthesisable RTL.

5.3.2 Hardware Implementation of the Correlator in Impulse CoDeveloper (Stage 3)

Initially, as a start-up point for creating a full-scale hardware correlation application a simple, two-channel correlator was implemented in Impulse CoDeveloper Application Manager. A simulation model running in Application Manager is demonstrated in Figure 5.5.

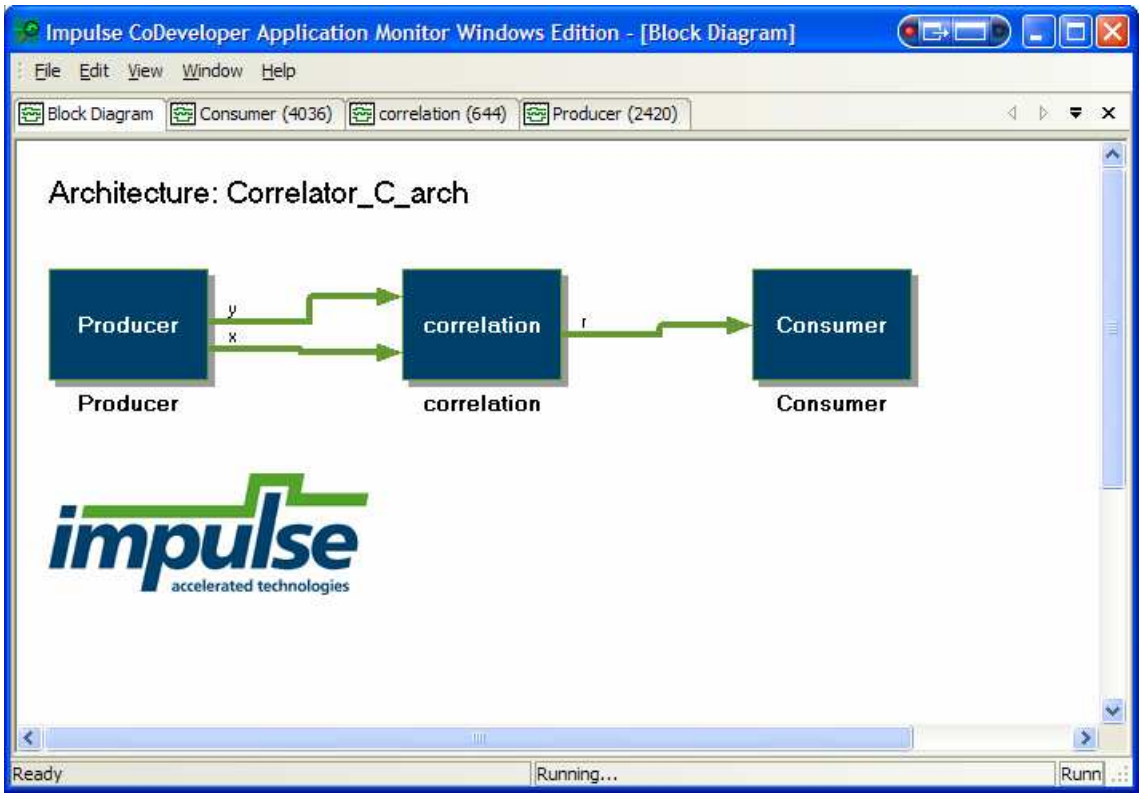


Figure 5.5. Simulation Model of Two-Channel Correlator Running in Impulse CoDeveloper Application Monitor

The developed model features two input streams (“x” and “y”) and one respective output (“r”). The main processing module, ie the hardware module with Impulse C code

converted from the software correlator, is called “correlation”. This particular module is converted to synthesisable Verilog HDL via “Generate HDL” command.

Impulse CoDeveloper streaming model also requires two additional entities: “Producer” and “Consumer”. The Producer process writes data into the actual, functional application process (to “correlation” in this case), whereas the Consumer accepts (reads) processed values from an output stream(s). Both processes are implemented in software module of Impulse CoDeveloper (see 4.3.2).

For the targeted algorithm of cross-correlation (Figure 5.4) the following manipulations with the code are required for Impulse C to generate efficient HDL. First of all, as long as the code relies on operations with arrays, loop unrolling will not be efficient in this case. According, to Impulse C support, the hardware can read at most two elements at a time from memories where arrays are stored. A more effective way of producing a parallelised HDL version of the algorithm is to introduce splitting of the arrays as shown in Figure 5.6.

```
for (lag = 0; lag < Nlag; lag++)
{
      tmp=nResult[lag];
      for(j=length/2; j < length; j+=4)
      {
      #pragma CO PIPELINE
            tmp+= nSample1[j]*nSample2[j-lag];
            tmp+= nSample1Copy1[j+1]*nSample2Copy1[j+1-lag];
            tmp+= nSample1Copy2[j+2]*nSample2Copy2[j+2-lag];
            tmp+= nSample1Copy3[j+3]*nSample2Copy3[j+3-lag];
      }
      nResult[lag]=tmp;
}
```

Figure 5.6. Introducing Splitting of the Arrays in Impulse C

`nSample1Copy*` and `nSample2Copy*` arrays are copies of arrays `nSample1` and `nSample2` with input samples. This alteration forces Impulse C to generate additional multipliers in the HDL and therefore increases the parallelisation of the algorithm. Additional temporary variable `tmp` is introduced to reduce multiple access to `nResult` array. For the example code in Figure 5.6, four copies of the input array are used, therefore, this code will use four multipliers on FPGA and, therefore, it should run four

times faster than the conventional code in Figure 5.4. Introducing additional copies of input arrays will increase the parallelism of the algorithm implemented in hardware.

The next step is to maximise the clock frequency for the generated HDL. This is achieved by manipulating with `stageDelay` parameter. The computations in an Impulse C process are broken down into stages. Each stage consists of a set of computations that can be performed in one cycle. Stage delay is defined by the maximum number of combinational delays or levels of logic within a given stage. Considerable stage delays may reduce the maximum operational frequency of the overall hardware design. `stageDelay` pragma introduces additional register stages in order to break down the longest propagation path in the hardware and, therefore, increase the potential maximum frequency. Introducing this parameter results in more pipeline stages but with increased overall throughput (Pellerin & Thibault, 2005). For the code in Figure 5.6 `stageDelay` will be introduced to break down additions of `tmp` variable into multiple stages:

```
#pragma CO PIPELINE
#pragma CO set stageDelay 32
      tmp1+= nSample1[j]*nSample2[j-lag];
      tmp1+= nSample1Copy1[j+1]*nSample2Copy1[j+1-lag];
      tmp1+= nSample1Copy2[j+2]*nSample2Copy2[j+2-lag];
      tmp1+= nSample1Copy3[j+3]*nSample2Copy3[j+3-lag];
      tmp = tmp1;
```

Figure 5.7. Using stageDelay Parameter in Impulse C

`tmp1` is introduced to aid with the breaking-down of additions into multiple stages. The definition of a right value for a `stageDelay` is aided by Pipeline Graph utility in Stage Master tool. This graphical tool plots values of `stageDelay` versus resulting theoretical operational frequency of the pipelined block (Effective Rate in Impulse C terms). Figure 5.8 shows an example Pipeline Graph plotted for 6-channel correlated version.

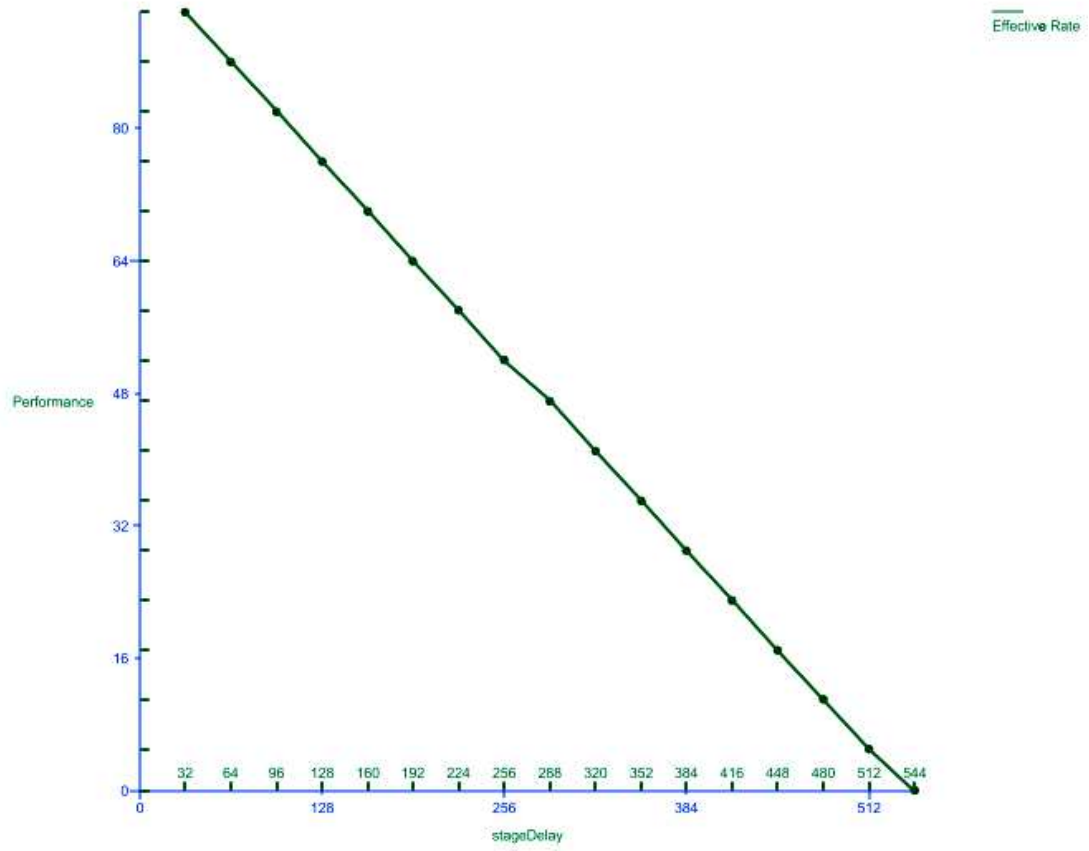


Figure 5.8. Pipeline Graph with stageDelay values for 6-Channel Correlator Design

From Figure 5.8 it is seen that `stageDelay` of 32 projects the maximum performance. It should be noted that the vendor's synthesis and place and route tools might introduce additional optimizations and path-breaking, thus affecting the final maximum operational clock of the pipelined block.

Using the configuration function of Impulse C, the developed two-channel correlator core was replicated the required number of times to facilitate the required number of channels. It should be noted that these replicated cores are also operating in parallel in hardware. 6-, 8-, 10-, 12-, and 16-channel versions of correlator were generated. For multi-channel correlation, signals have to be re-utilised in computations. For example, for 4-channel correlation of signals x1, x2, x3, and x4 the output cross products will be: x1×x2, x1×x3, x1×x4, x2×x3, x2×x4, and x3×x4. In this example each signal participates in computation three times (for three output cross-products). This issue is important in Impulse C since data streams have one-to-one connectivity and cannot be connected to multiple correlation cores.

To circumvent this problem a simple demultiplexor process was implemented in Impulse C hardware module. The purpose of this process is merely to replicate an input stream required a number of times, so that each of those copies of the input stream will

be connected to precisely one correlation core. An example of 6-channel correlator top-level entity with implemented demultiplexors is given in Figure 5.9. Respective listings of software and hardware modules and the include file for a 6-channel correlator are given in Appendix A5. The idea of implementing cross-correlation algorithm in an Impulse C environment might be derived from this design. Other versions of correlator design (8-, 10-, 12-, and 16-channels) are, in fact, the same and are presented only on the CD enclosed with this thesis. They differ only in the Impulse configuration function, which instantiates and connects hardware and software processes in a specified manner.

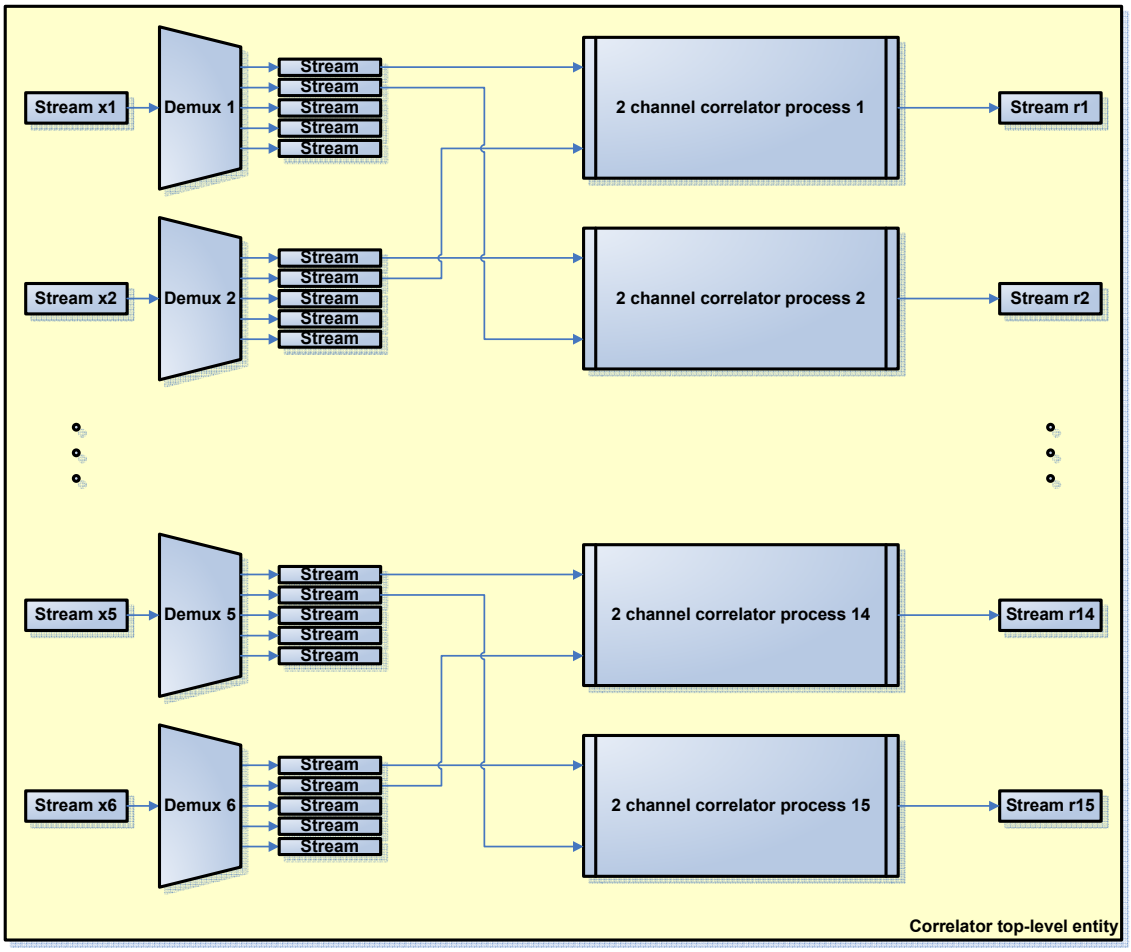


Figure 5.9. Top-Level Entity of the 6-Channel Correlator with Implemented Demultiplexors

The reported delay introduced by these demultiplexors is one clock cycle and, therefore, its influence on the overall design's performance can be negated.

As mentioned in 4.3.2, only one concurrent software process is supported in the exported Impulse C software module on XD1000. This effectively means, that providing data to a hardware process(es) from the software part of Impulse C design has to be done in a sequential manner or some sort of synchronisation mechanism has to be provided. However, available realisation of Impulse C software-to-hardware

synchronisation tools on XD1000 PSP has certain shortcomings. This issue is discussed in more detail in 7.1.1.

To test Impulse C capabilities of generating accelerated HDL code from ANSI C input, the execution time of correlator design will be measured. This time will be then compared with execution software implementation of the correlator algorithm run on a conventional PC. The time is measured from a software module with Linux `gettimeofday` functions (bits of code measuring time are commented on in Appendix A5 to maintain compatibility when simulated in a Windows environment). These functions measure time with an accuracy of nanoseconds. Respective results are presented in 6.1 and discussed in 7.1.1.

After the performance of an Impulse C project had been verified, it was exported into ready-for-synthesis Quartus II project using the "Export Generated Hardware (HDL)" feature. Software module of the Impulse C project is exported using "Export Generated Software" feature. The Quartus II project is then compiled and the received bit stream programs XD1000 target server. Exported software project is transferred to XD1000 where it can be compiled by a standard GCC compiler and executed.

5.4 I/O Framework (Stage 4 and Stage 5)

As established in 3.1.2, implementation of I/O interfaces in FPGA-based design can be a challenging and time-consuming problem. Hence, it is highly preferable to re-use already available and working I/O designs for PCIe Stratix II GX board to reduce the development time. In the given project, PCI Express to DDR2 SDRAM Reference Design from Altera Corporation will be used as a foundation for the I/O framework.

5.4.1 PCI Express to DDR2 SDRAM Reference Design from Altera Corporation

The block diagram of PCI Express to DDR2 SDRAM Reference Design from Altera Corporation reference design is given in Figure 5.10.

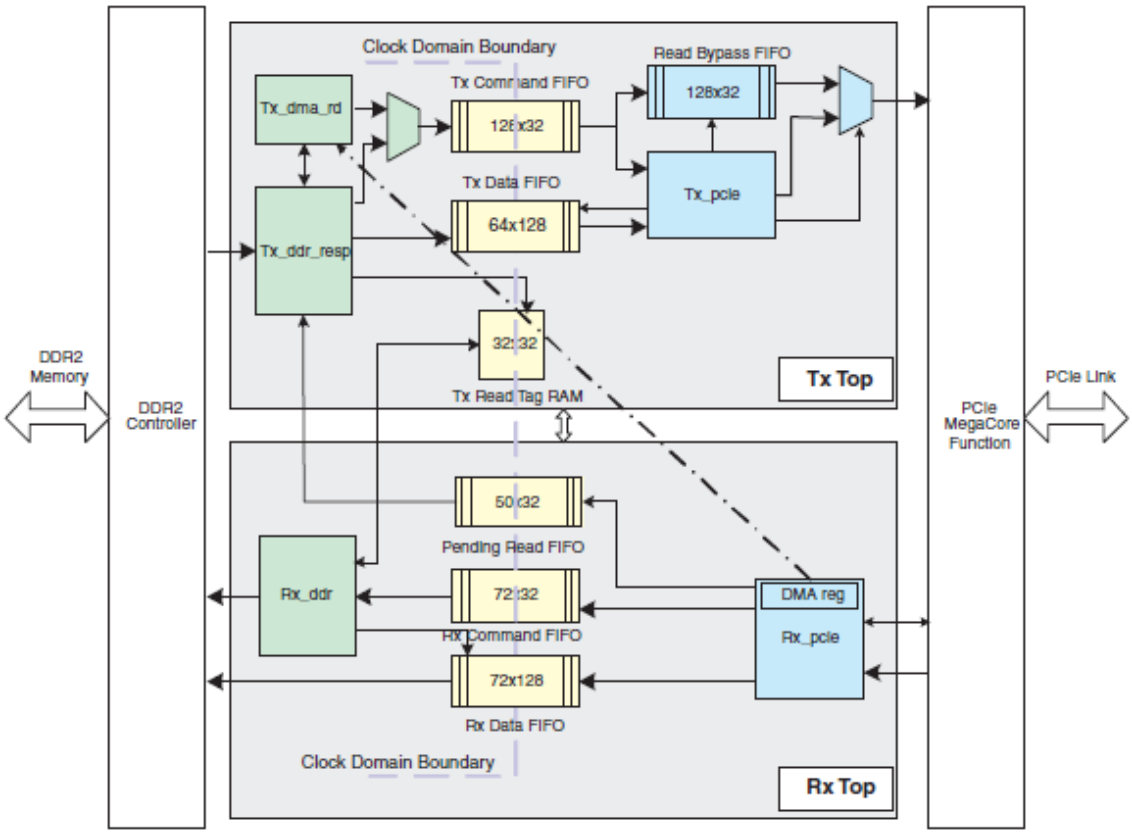


Figure 5.10. PCI Express to DDR2 SDRAM Reference Design Block Diagram (Altera Corporation, 2006)

This design uses the Stratix II GX PCI Express Development Kit as a hardware platform and features Altera's PCI Express MegaCore function core which instantiates PCIe interface in the maximum available configuration for the board – ×8. The Reference Design provides an example interface between the Altera PCIe MegaCore function and the Altera's DDR2 SDRAM Controller MegaCore function that enables access to external 64-bit, 256 MB DDR2 SDRAM memory through the PCIe bus. Hence, the design operates in two clock domains – clock domain of PCIe core and clock domain of DDR2 Controller. Altera's PCI Express to DDR2 SDRAM Reference Design also demonstrates an example of a typical user application (GUI-based Windows Application) that interfaces to the system side of the Altera PCIe MegaCore function. The GUI application of the reference design performs read and write operations to the onboard SDRAM vie ×8 PCIe interface. This application has only demonstrational capabilities: the types of data for transfer are pre-defined (zeroes, ones, random, etc), the maximum size of transfer is limited to 4,096 bytes, etc.

This reference design does employ actual DMA algorithm, although DMA logic is present in the design files and demonstrational GUI software has "DMA Read" and

"DMA Write" options. When these options are used, common CPU-polling transactions are performed.

Nevertheless, this design was used as a starting point for I/O framework development.

5.4.2 Developed I/O Framework (Stage 4)

The framework design above is re-used in the given project with necessary amendments and modifications. Figure 5.11 shows the full block diagram of the implemented I/O framework.

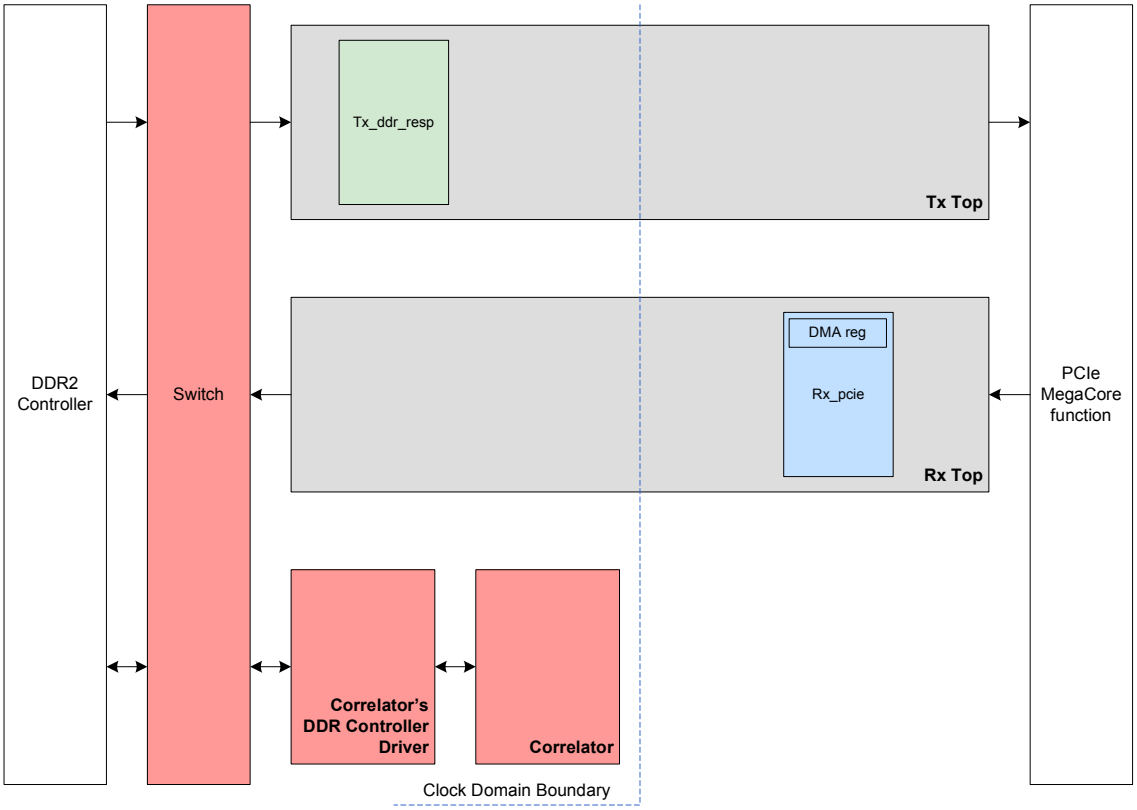


Figure 5.11. I/O Framework Block Diagram

Blocks in red represent logic introduced into the original reference design. Coloured blocks tx_ddr_resp and rx_pcie in the original design logic (blocks in grey) are the only blocks modified in the original design. Their full code listings are given in Appendix A6 with introduced changes indicated by respective code comments.

The main idea of the implemented I/O framework is to preserve as much as possible of the reference design's logic while providing necessary interfacing of the correlator to the onboard DDR2 SDRAM memory. All communications from the system's side of PCIe interface are preserved, which allows to perform read and write

operations from PC's side. Furthermore, all of the reference design interface signals to DDR2 Controller are connected via the switch block. The switch is also connected to the Correlator's DDR Controller Driver. Hence, the switch performs merely controlling functions connecting either the original reference design or the correlator. The switch is necessary as only one instance can operate with the DDR2 Controller at any one moment in time.

The switch is controlled from the PC via the DMA Control register. Although the reference design features DMA logic, the DMA mechanism is not supported – although DMA-initiated transactions can be issued from a demonstrational GUI application, they are implemented by means of constant polling of the CPU. Hence, the DMA Register is used in this implementation entirely for control purposes: by asserting reserved 20^{th} bit in the DMA Register the switch connects the Correlator's DDR Controller Driver to the DDR2 Controller, thus allowing the correlator to work with the onboard memory. The code of the switch module is given in Appendix A7.

The Correlator's DDR Controller Driver reads the onboard memory and feeds the correlator with unprocessed data. Once the correlator finishes processing, the driver reads processed data from the correlator and writes it to the SDRAM. After this, the switch connects the original design (grey block Rx Top and Tx Top in Figure 5.11) to the DDR Controller, restoring the software application's control over the SDRAM. To test and debug the Correlator's DDR Controller Driver the "Interfacing DDR2 SDRAM with Stratix II, Stratix II GX, and Arria GX Devices" reference design was used (Altera Corporation, 2007b). This design featured a demonstrational DDR driver, which was modified to suit the desired functionality of the Correlator's DDR Controller Driver. Besides, this reference design features simulation ModelSim model for verification of the design performance. ModelSim provides a comprehensive software simulation and debug environment for Verilog and VHDL designs. Software simulation of ModelSim exposes implicit hardware state in the FPGA (see challenges in FPGA programming in 3.1.2) and reduces hardware debugging efforts. The code of the Correlator's DDR Controller Driver is listed in Appendix A8.

Simultaneously with connecting the switch to the application interfaces, the application (Correlator block in Figure 5.11) itself is enabled and starts acquiring data from the DDR2 SDRAM and performs processing according to its algorithm. Once the processing is finished and output data is recorded to the memory, the application returns control to the original design (a switch connects the original design's interfaces to

DDR2 SDRAM) and also asserts 25th bit in the DMA Register (also reserved bit), which is checked by the Software Control Application and serves as a "processing complete" flag.

5.4.3 Software Control Application (Stage 5)

The aforementioned software console application was developed in Jungo WinDriver PCI/PCI Express/PCMCIA development tool, which comes with the board. The application is based on the diagnostic application utility for Altera's PCIe-featuring boards – pci_dev_kit. The code of this application was generated by WinDriver Wizard and the whole utility is supplied as an example of accessing Altera hardware by provided WinDriver functions.

The current version of software control application supports writing and reading data to and from the onboard SDRAM memory via instantiated PCIe ×8 link. There are several supported write and read functions in pci_dev_kit diagnostic applications:

- `ALTERA_WriteByte` – writes 8 bits of data;
- `ALTERA_WriteWord` – writes 16 bits of data;
- `ALTERA_WriteDword` – writes 32 bits of data;
- `ALTERA_ReadByte` – reads 8 bits of data;
- `ALTERA_ReadWord` – reads 16 bits of data;
- `ALTERA_ReadDword` – reads 32 bits of data.

Figure 5.12 shows example calls of `ALTERA_WriteWord` and `ALTERA_ReadDword` functions.

```
ALTERA_WriteWord(hALTERA, ad_sp, j, (WORD)file1[i]);
data = ALTERA_ReadDword(hALTERA, ALTERA_AD_BAR2, 0xC);
```

Figure 5.12. Example Calls of Write and Read Functions to Onboard SDRAM Memory

`ALTERA_WriteWord` function performs write of 8-bit value `file1[i]` to offset `j` at memory address space `ad_sp` in device specified by handler `hALTERA`. Similarly, `ALTERA_ReadDword` reads 32-bit value from offset `0xC` in memory address space `ALTERA_AD_BAR2` and returns the read value to the data variable. The reference design,

which is used as a foundation for the developed I/O framework, supports the following memory address spaces:

Table 5.2. Memory Address Space in PCI Express to DDR2 SDRAM Reference Design (Altera Corporation, 2006)

Memory Region	Block Size	Memory Type	Description
BAR0 & BAR1	16 MByte	64 bit, prefetchable	16 MByte DDR2 memory range capable of supporting 24 bits of address bus
BAR2	4 KBytes	32 bit, non-prefetchable	Internal reference design DMA configuration registers

Data exchange between the computer and onboard SDRAM memory is carried out via BAR0 or BAR1. Control and configuration commands operate with BAR2. For example, the switch module is controlled by asserting reserved 20th bit at offset 0xC of BAR2 (the DMA Register) – see 5.4.2 for more details.

Although software control application supports DMA requests and the reference design has implemented DMA mechanism (DMA registers, DMA control mechanisms, etc), DMA read and write operations are not supported. Consequently, the developed software control application inherits this limitation. The impact of this shortcoming is discussed in 7.1.2.

The current version of the software control application was developed targeting integrated Impulse C correlator design. Nevertheless, the I/O framework and software control application as a part of this framework can be adapted and modified to accommodate other desired applications.

The only file modified from original pci_dev_kit application is alt_pcidiasg.c. Its listing is given in Appendix A9.

5.5 Chapter Summary

This chapter describes how the implementation process of this work was undertaken. Visualisation of the project stages, development platforms and software tools applied at these respective stages are presented in Figure 5.1. Initial trial correlator design setting design approaches are presented, which are followed by conditions and reservations accepted for the correlation implementation in this work.

The remainder of this chapter consists of two major parts: the correlator implementation and the I/O framework. The correlation implementation part introduces experiences of extracting parallelism and achieving accelerated HDL performance of correlation algorithm in Impulse C. The I/O framework part describes how PCI Express to DDR2 SDRAM Reference Design was used as a foundation for the framework and also highlights the creation of software control application in the Jungo WinDriver PCI/PCI Express/PCMCIA development tool.

The next chapter presents the results obtained from the implementations of this chapter.

CHAPTER 6

Results

There's two possible outcomes: if the result confirms the hypothesis, then you've made a discovery. If the result is contrary to the hypothesis, then you've made a discovery.
— Enrico Fermi

This chapter presents results obtained from the implementation stages of this work: the correlator hardware design on XD1000 system from Stage 3 and the I/O framework for the high-performance hybrid DSP system with software control application from Stage 4 and 5.

6.1 Correlator Design (Stage 3)

This section will present performance results obtained from the execution of the reference program on a conventional PC (Stage 3), which is referred to as software execution or “SW”, and from the execution of generated, synthesised and programmed Impulse C correlator projects on the XD1000 development system (Stage 2), which is referred to as hardware execution or “HW”. As established in 5.2.2, a correlator with various numbers of channels (6, 8, 10, 12, and 16) has been designed and tested in both software and hardware.

According to Equation (2.2), depending on the number of input channels, a correlator produces a different number of unique outputs or cross-products. Table 6.1 gives the number of cross-products for the selected range of input channels.

Table 6.1. Number of Output Cross-Products Depending on the Number of Input Channels

# of input channels	# of output cross-products
6	30
8	56
10	90
11	110
12	132
16	240

The problem size demonstrated in Table 6.1 has an impact on the logic utilisation of the EP2S180F150C3 chip. For 6-, 8-, 10-, and 11-channels the logic utilisation was 31, 51, 84 and 97% respectively (with eight copies of input arrays introduced for array splitting – see 5.3.1). The 12-channel correlator design exhausted all the available logic registers in XD1000's EP2S180 device: it required 159,964 logic registers whereas the FPGA contains only 143,520. Therefore, hardware implementations were limited to a maximum 11-channels.

Hardware implementations were simulated using an Impulse C Stage Master Debugger, which was discussed in 4.3.2. This simulation determined the precise number of clock cycles required for pure processing only, ie how many cycles elapsed since the first sample arrived in computational cores until the very last output sample is recorded.

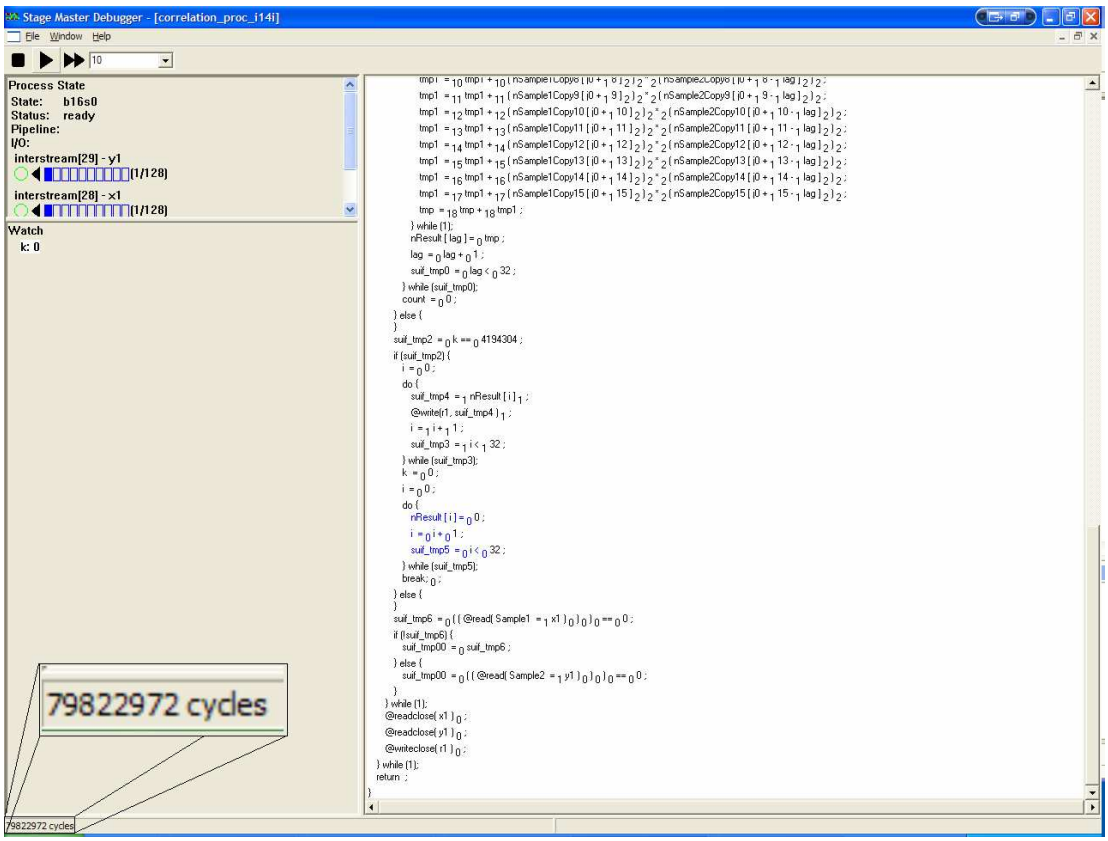


Figure 6.1. Simulation of Hardware Implementation in Stage Master Debugger

The zoomed section in Figure 6.1 shows the number of elapsed clock cycles for full processing. The generated computational cores are executed in parallel so the number of elapsed clock cycles in simulation will be the same for any number of channels. By dividing the number of elapsed clock cycles by the operating frequency of the FPGA (see 4.2.2) an estimated theoretical execution time can obtained.

The reference program from Stage 3 was executed on a conventional PC with the following configuration: Intel Pentium D 3.4 GHz, 1 GB of RAM, Windows XP Professional with SP2.

For software and hardware implementations the design was run three time and an averaged value of execution time was recorded. The actual measured execution time for software and hardware implementations along with simulation execution time for hardware are plotted in Figure 6.2.

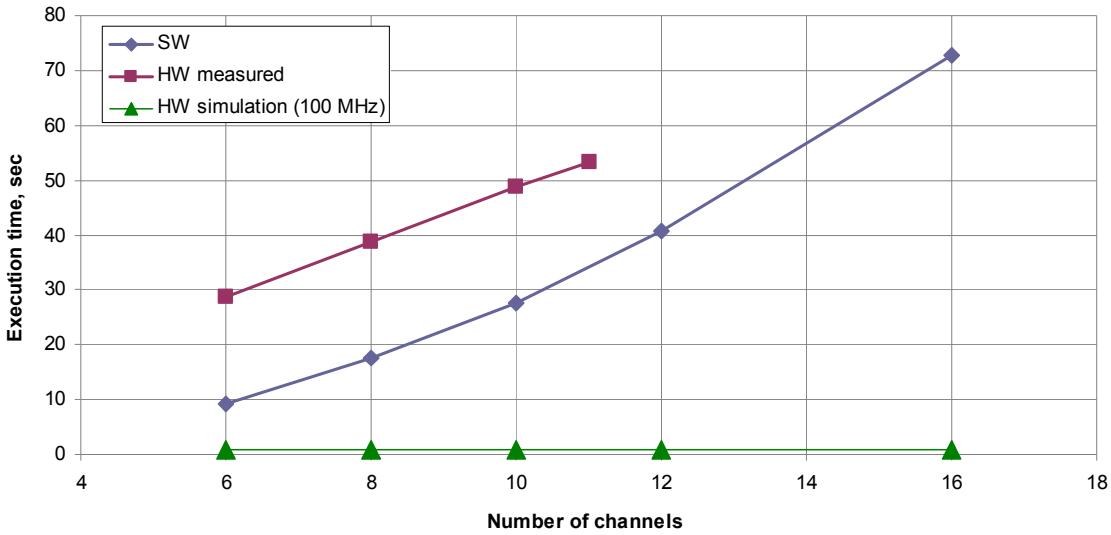


Figure 6.2. Performance Results of Software and Hardware Implementations

Table 6.2 presents data used to plot graph in Figure 6.2.

Table 6.2. Performance Results Data of Software and Hardware Implementations

Number of channels		6	8	10	11	12	16
Number of cross-products		30	56	90	110	132	240
Execution time, sec	SW	9.28	17.50	27.72	32.1	40.68	72.73
	HW measured	28.80	38.62	48.87	53.15		
	HW simulation (100 MHz)	0.80	0.80	0.80	0.80	0.80	0.80
	HW simulation (400 MHz)	0.20	0.20	0.20	0.20	0.20	0.20

Table 6.2 also features data for hardware simulation when the FPGA operational frequency is 400 MHz (not plotted in Figure 6.2). The current version of XD1000 PSP by default limits the FPGA frequency to 100 MHz. However, the employed Stratix II device supports frequencies of up to 400 MHz and operational frequency can be changed by editing PLL properties.

These results are discussed in 7.1.

6.2 I/O Framework with Software Control Application (Stage 4 and Stage 5)

Another outcome of this work is the operational I/O framework for CPU+FPGA architecture. As an estimation test of the developed framework a 6-channel correlator generated from an Impulse CoDeveloper was programmed into the PCI Express Development Kit Stratix II GX edition. Software control application was used to write 4,096 bytes of data into onboard SDRAM memory. Then the application triggered the correlator design and read the processed values from the memory after the "processing complete" flag (25th bit of the DMA Register) is asserted (see 5.4.2). A sample run of the Software Control Program for the 6-channel correlator processing 4,096 8-bit samples is shown in Figure 6.3:

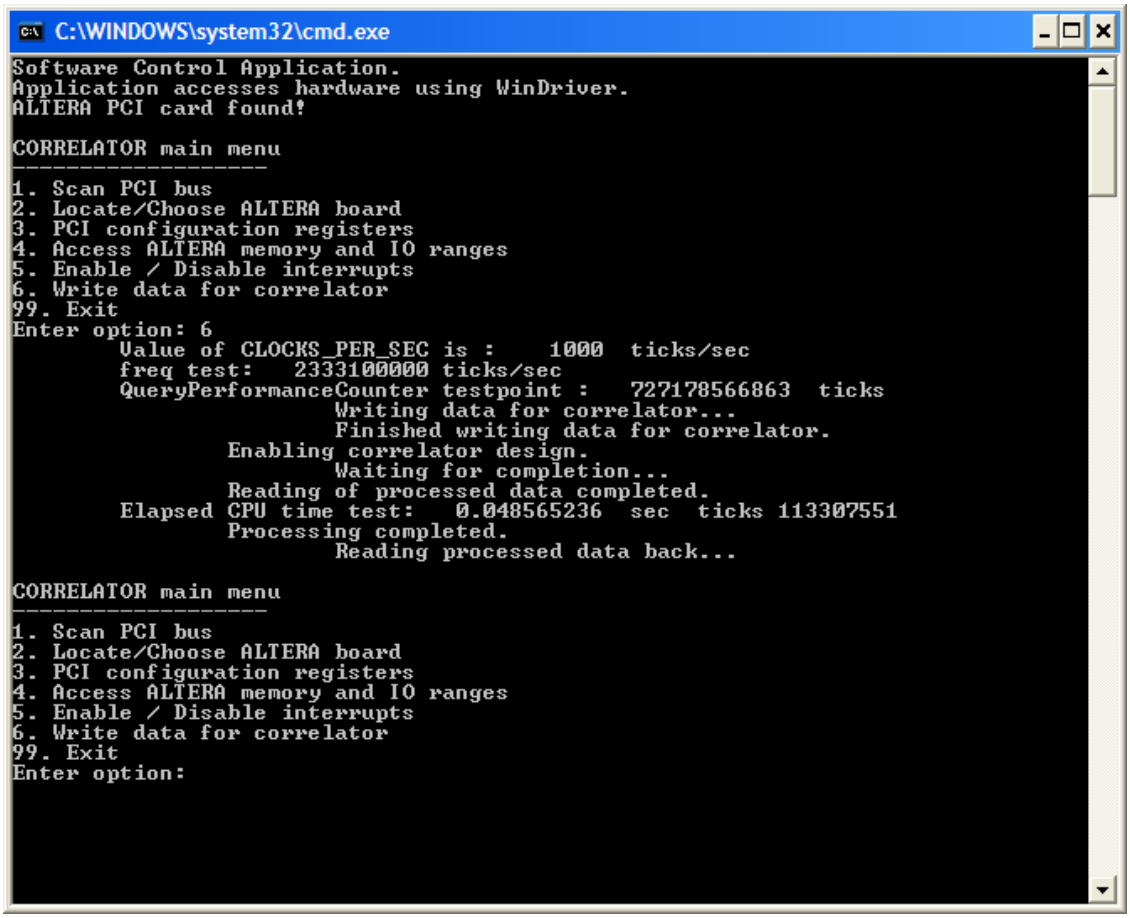


Figure 6.3. Sample Run of Software Control Application

CHAPTER 7

Discussion

For everything you have missed, you have gained something else, and for everything you gain, you lose something else.
— Ralph Waldo Emerson

The structure of this chapter is as follows: first, discussion of the obtained results is given, which is followed by suggestions of alleviating known shortcomings and future developments. The chapter concludes with an overall summary.

7.1 Discussions

This work delivers two main outcomes:

1. Multi-channel cross-correlator design working in a CPU+FPGA architecture and developed with new top-down design methodology (Figure 4.7).
2. I/O framework for CPU+FPGA architecture with software control application.

All research objectives (see 1.2) are achieved, satisfied and are covered by outcomes of this project. The following sections group the discussions for these respective outcomes.

7.1.1 Correlator Design

As can be seen from Figure 6.2, hardware design simulation with operational frequency 100 MHz achieves speedups from ×10 to ×90 for a different number of channels over software implementation. Table 7.1 summarises hardware speed-ups versus software for 100 MHz and 400 MHz operational frequencies.

Table 7.1. Achieved Simulation Speed-ups

Number of channels	6	8	10	11	12	16
Number of cross-products	30	56	90	110	132	240
Speed-ups (100 MHz)	11.63	21.92	34.72	40.20	50.96	91.11
Speed-ups (400 MHz)	46.52	87.68	138.88	160.79	203.85	364.44

These speed-ups in hardware implementation are "obscured" by the polling streaming approach applied in XD1000 PSP. This issue will be discussed in more detail. In limitations of XD1000 PSP listed in 4.2.2, it was stated that software-hardware communication via streams is not DMA. Instead, input-output is CPU polling and yields only 2 MB/sec of the supported 400 MB/sec for each link of the HyperTransport interface (full HT, however, yields to 3.2 GB/sec). This is also supported by the fact that when different parallelisation effort (different granularity of array splitting) is applied to a correlator with a fixed number of input channels, the execution time remains the same. Theoretically, different parallelisation of the same correlator design should produce different performance results, which are not visible due to the slow I/O communication.

One option to alleviate this shortcoming of slow streaming interfaces in XD1000 implementation is to use shared memory communication. In this approach, input and output data is transferred across SRAM (DDR2 SDRAM is not supported) memory available on XD1000 co-processor module. The shared memory yields 800 MB/sec of bandwidth and is likely to overcome the issue of slow communication for the correlator design. However, the implemented streaming approach is more natural for DSP applications and, therefore ,was preserved for future developments of the project.

Nevertheless, even shared memory approach with greater bandwidth has challenges when it comes to actual implementation. Working with shared memory from both software and hardware processes requires some sort of scheduling or synchronisation mechanisms. Impulse C features semaphores (co_semaphore), which serve precisely this purpose – to perform one-to-many process synchronisation. Unfortunately, currently semaphores are supported only for hardware processes' synchronisation and are not callable from software processes. This leaves only one remaining option for synchronisation – Impulse C signals (co_signal). However, these signals are one-to-one synchronisation and therefore are impractical to perform control over shared memory access for applications featuring dozens of processes. Therefore, even though shared memory communication offers higher bandwidth than streaming

communication, it is unfeasible for many applications including the targeted one in this project.

In addition, as mentioned earlier, no concurrent software processes are currently supported in XD1000 PSP. This implies that communication from the software process to the inherently parallel hardware process implemented in an FPGA accelerator has to be done in a sequential manner, which limits the parallelisation degree of the whole system.

The overall efforts required to implement a scalable, multi-channel correlation design using high-level FPGA programming in Stage 2 are significantly less than using traditional FPGA development tools in Stage 1. Moreover, the correlator design of Stage 1 required more development time but has less functionally than the correlator developed in Stage 2, which is more flexible and scalable. The process of developing the correlator in Impulse C in Stage 2 required little to no HDL design techniques: clock and parallel execution had to be considered. However, no low-level debugging tools (signal analyzers) were used whatsoever.

The fact that Impulse C can actually generate accelerated HDL from C input demonstrates the potential of this tool. Admittedly, certain manipulations, uncommon for conventional programming, had to be performed with the C code to produce efficiently parallelised hardware design. For the targeted correlator design, these manipulations included array splitting, pipelining, and introducing stage delays. In fact, to generate HDL designs with reasonable speed-ups from high-level FPGA programming tools, a knowledge and practical understanding of hardware operation are still required. While there is yet no “green button” solution to generate final and complete hardware designs from entirely software algorithm implementations, consideration of hardware implementation execution and understanding of parallel and clock concepts are required. This issue is discussed more in 7.3.

It should be also noted that, current implementation of input arrays splitting requires replicated copies of the same input arrays, which results in increased logic utilisation. A more practical way of introducing parallelism while avoiding FPGA fabric waste, is copying input data into smaller sub-arrays each storing a separate portion of input data samples. Such approach will require respective changes to cross-correlation algorithm (Figure 5.4).

While the results demonstrate the advantage of using reconfigurable hardware for performance enhancement of the DSP application, several key issues should be taken

into account. Firstly, the C-to-HDL conversion is not as sophisticated and efficient as manually created ones yet. Logic utilisation of automatically generated designs is an important issue, which should be taken in account when working with C-to-HDL tools. As reported in 6.1, 12-channel correlator could not fit into capacious Stratix II EP2S180F150C3 chip. Since little to no control tools over logic utilisation exist when HLP is applied for FPGA programming, the issue of effective resource utilisation is one of the major concerns when working with C-to-HDL compilers. The only option so far to fit a large-scale Impulse C design into the targeted FPGA is to use a trial-end-error approach and refine the input C-code: reduce number of used variables, limit buffers' and arrays' sizes, etc and then attempt to place and to route to see if the refined design fits the targeted device.

Due to limitations of Impulse C parser, certain parts of configuration function (4.3.2) had to be created manually or with the help of scripts, which generated configuration function code. For example, the parser of configuration function cannot handle manipulation of arrays in complex for-loops containing more than one index. Such loops were utilised for interconnecting copies of input streams after demultiplexors. Due to similar deficiencies of the parser, the exported module of the correlator project is not generated correctly and, therefore, has to be modified manually (providing correct names of the copied streams in the exported file co_init.c).

Another encountered limitation of Impulse C is that the hardware process can take only up to 32 arguments, ie 32 interfaces in total. This limitation, however, did not affect this implementation where a process with the most interfaces is Producer, which arranges input channels and feeds the computational cores.

It has to be mentioned that the C source code of the reference correlation program (Stage 3) is not aimed to calculate cross-correlation functions with maximum efficiency. Rather it was created as a reference point and a foundation for Impulse C implementations. The purpose of the reference program is to prototype a situation when a working algorithm written in HLL is adapted to a hardware implementation by a C-to-HDL compiler. Therefore, another option of increasing the processing performance is to elaborate the implemented algorithm.

A similar concept applies to model signals. The selected signals are entirely test signals introduced only to test and verify correlation algorithm. Nevertheless, they belong to 128 MHz bandwidth, which is traditional for many DSP applications and are sampled with 6-bits to prototype data coming from ADCs.

7.1.2 I/O Framework

The I/O framework presented in this work is feasible to accommodate both correlator designs generated from Impulse CoDeveloper as well as manually developed ones. The framework can accommodate other DSP applications with streaming and a memory-buffered approach with few or no changes at all required.

The approach of re-using the original reference design was selected due to the two primary reasons:

1. To fit into the development timeframe by re-using as much of the existing and configured I/O communication as possible, thus addressing the FPGA's I/O interfaces challenge discussed in 3.1.2.
2. The possibility to implement demultiplexing approach in wideband correlators. This approach is explained in the following paragraph.

The sampling bit rates of modern ADCs are an order of magnitude faster than operational rates of VLSI integrated circuits (few hundreds Mbit/s) which are widely used for large-scale correlators. For this purpose a demultiplexing approach is applied for lag correlators – each sample output is divided into *n* streams with *n* contiguous samples all going to a different stream. The result is obtained by cross-correlating each stream of one signal with every stream of the other signal (Thompson et al., 2001a). For example, the MMA correlator (Escoffier, 1997) features a workflow where demultiplexed bit streams work with a large RAM. The output streams from a sampler are recorded in *n* blocks each containing a contiguous data as sampled. The corresponding blocks of data are then read out and cross-correlated by the correlator. The output is recorded into the RAM as well.

Hence, the current implementation employs the same demultiplexing approach where onboard SDRAM is used as an intermediate buffer where the demultiplexed unprocessed data and output results are stored. The demultiplexing approach allows processing of substantial amounts of data characteristic for wideband correlators and is very practical for the selected framework. This demultiplexing approach should not be confused with demultiplexors in Impulse CoDeveloper correlator design (5.3.2), which had to be introduced to alleviate one-to-one connectivity of Impulse C streams.

Similar to the correlator design implementation on XD1000, the I/O framework developed for the high-performance hybrid DSP system (3.2) lacks DMA interrupts. The I/O framework inherited the polling mechanism for reading and writing data from

and to the board's SDRAM from the PCI Express to DDR2 SDRAM Reference Design. A more effective way to perform memory read/write is to use DMA, which does not involve the CPU for the whole period of interaction as opposed to Programmed Input/Output (PIO).

Presently the CPU's involvement in the developed Software Control Application comes down entirely to administrative and control functions: streaming data in the actual processing system implemented in the FPGA, polling for and fetching processed data back. Whereas, to highlight the actual benefits of CPU+FPGA architecture, the CPU can be involved in a shared or selective computation, eg in completely un-parallelisable computations which cannot be accelerated in reconfigurable hardware.

In addition, implementation of the data exchange interfaces (PCIe core, DDR2 SDRAM controller) on the FPGA was taken from the reference designs and might be enhanced. For example, DDR2 core is clocked with a conservative clock of PCIe core of 250 MHz, whereas the reference design features a clock domain boundary and, therefore , the DDR2 controller can be theoretically clocked with a supported 333MHz. These issues leave significant room for further sophistication and enhancement of the targeted correlation applications.

To develop, test and debug the framework approximately ~6 months of work were spent. Traditional development tools were applied at this stage – HDL coding and the SignalTap Analyzer monitoring tool for debugging. First, this indicates that traditional hardware design methodology is challenging to work with and, therefore, time-consuming. Second, it highlights that even if reconfigurable hardware features high-speed interfaces beneficial for a certain application, to employ these interfaces efficiently in a real-life implementation often comes down to a time-consuming task. This predominantly happens due to the necessity to develop basic input-output interfaces practically from scratch – see 7.3 for more details.

7.2 Future Work

As stated in 3.2, the full projected system is highly flexible. The number of available interfaces on the PCI Express development board allows vast opportunities for interconnection and makes the proposed system exceptionally scalable. The fact that the proposed system uses a commodity PC as a platform implies even far bigger perspectives, as the system can be scaled at the PC's cost, the cost of the FPGA PCIe

board and the cost of the ADC kit, which even added up together still place the proposed CPU+FPGA framework as an actual challenger to complex and expensive high-profile DSP solutions (eg BEE2). The average cost of the proposed system consists of the cost of the PCIe development board – $1,500 USD (academic price) and cost of a commercially- available PC – $1,000-$2,500 USD. Whereas the indicated cost of the BEE2 system is $3,000 to $5,000 USD without the cost of FPGAs. Moreover, the proposed system (Figure 3.7) is not limited to cross-correlation only: it can also accommodate a range of high-performance DSP applications.

In the current implementation of the I/O framework, the data is written to and read back from the onboard memory for buffering, whereas potentially the board has all the interfaces to perform real-time processing and implement an entirely streaming approach. The board has high-speed Mezzanine connectors which are routed to the transceivers inside Stratix II GX chip. This option allows direct, intermediate data acquisition (eg from ADCs) and further streaming of it into the chip for immediate processing. Furthermore, the output, processed data can be streamed via any of the available interfaces: PCIe, Gigabit Ethernet or SFP. The next section gives an example of how a high-performance DSP application performing real-time processing can be implemented on the developed framework.

As stated earlier, the I/O framework does not support DMA, which for real-life processing tasks can become a serious reason for performance degradation. The PCI Express to DDR2 SDRAM Reference Design (5.4.1) has embedded DMA logic which, however, is not used in the actual operation. As for the developed software control application, it already supports DMA interrupts. Enabling DMA support in the reference design is recommended in case of employing the framework in real-life high-speed processing and, therefore, considered as one of the future development stages.

For the correlator design, the primary goal was to achieve a speed-up in the computational core of the correlation algorithm on hardware platform. Therefore, in case of the further development of the correlator design a speed-up might be searched outside of the correlation computation loops – more careful clock cycle considerations of input and output streaming interfaces, eliminating or merging variables manipulation stages, etc.

Besides, as it was mentioned the implemented code of cross-correlation (Figure 5.4) was not developed with the intention of performing cross-correlation with maximum efficiency. Therefore, the cross-correlation algorithm itself can be improved.

Additionally, in case the correlator design is applied for correlation calculation in practical applications (eg for radio-astronomical applications), an FX type of correlation might be considered. Due to the fact that in this correlation Fourier transform is applied *before* calculating cross-products, it offers some advantage in the required number of cross-products calculations, as opposed to XF correlators where Fourier transform is applied after cross-products are calculated.

The XD1000 implementation of the correlator can be enhanced by employing shared memory interfaces between software and hardware processes. This will allow to overcome the shortcoming of the slow streaming communication of the XD1000 PSP. However, as stated in 7.1.1, to implement the shared memory approach a synchronisation mechanism is required between the communicating processes. Currently, such a synchronisation utility is not feasible to implement with the supported tools. Support of such synchronisation tools will theoretically increase communication throughput between processes and, therefore, the overall XD1000 performance.

7.3 Summary

Some parts of the work presented in this thesis have been published in several sources: (Leonov & Kitaev, 2007) and most recently, (Leonov & Kitaev, 2008).

The proposed approach of simultaneous involvement of the CPU and the FPGA for correlation in this thesis can be expanded to other DSP applications, such as image-processing, telecommunication, cryptography, provided that the data input-output interfaces are fast, well-tested and reliable. As stated before, FPGA has no knowledge about any I/O interfaces before it is configured, whereas for any DSP application input-output throughput is one of the critical questions in achieving top performance. Thus, although CPU+FPGA-based computing can deliver a significant increase in performance for a number of applications, the current state of FPGA development requires a substantial amount of expertise and design efforts to achieve the respective speed-up.

The new top-down design flow (Figure 4.7) applied in this work has demonstrated its viability for developing DSP applications in hybrid CPU+FPGA architecture. Applying HLL for FPGA programming enables production of high-speed applications working in a reconfigurable computing environment with development time conservation in comparison with traditional hardware development workflow. However, certain challenges still exist in the field of C-to-HDL compilers. Apart from employing

naturally sequential languages for parallel programming, implementation of I/O interfaces (particularly for such I/O-sensitive applications as DSP applications) remains one of the most considerable challenges for the compilers. Figure 7.1 outlines application implementation techniques for reconfigurable computing.

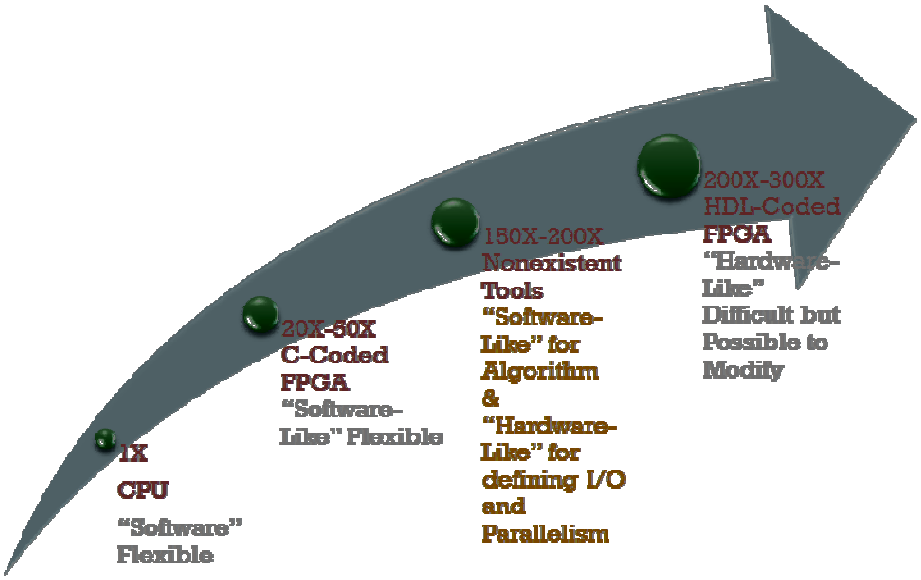


Figure 7.1. Application Implementation Techniques (Kitaev & Molteno, 2008)

According to Figure 7.1, for currently existing high-level FPGA programming tools speed-ups of ×20 to ×50 are achievable: similar speed-ups were achieved in this work with an Impulse CoDeveloper (see 6.1). It has to be acknowledged that C-coded hardware designs are not as efficient in terms of achieving performance acceleration as HDL-coded ones. However, there is a way to bridge the gap between them. It is envisioned that current nonexistent hybrid tools with "software-like" approach of implementing algorithmical steps and "hardware-like" techniques of determining I/O interfaces and parallelism will be capable of creating designs achieving speed-ups comparable to manually-created ones. The experiences of this work comply with this hypothesis – cross-correlation algorithm implementation with Impulse C means took the small portion of overall time and efforts, whilst parallelism extraction and I/O framework development required the most attention and multiple stages. Therefore, the output of this thesis conforms to the idea that such hybrid tools are necessary and require attention from the DSP research community.

Another interesting research direction is to have an entity, which will automatically decide which part of a DSP algorithm is optimal to execute in hardware

and which part in software (Galanis, Milidonis, Theodoridis, Soudris, & Goutis, 2007). The data flow of this idea is given in Figure 7.2.

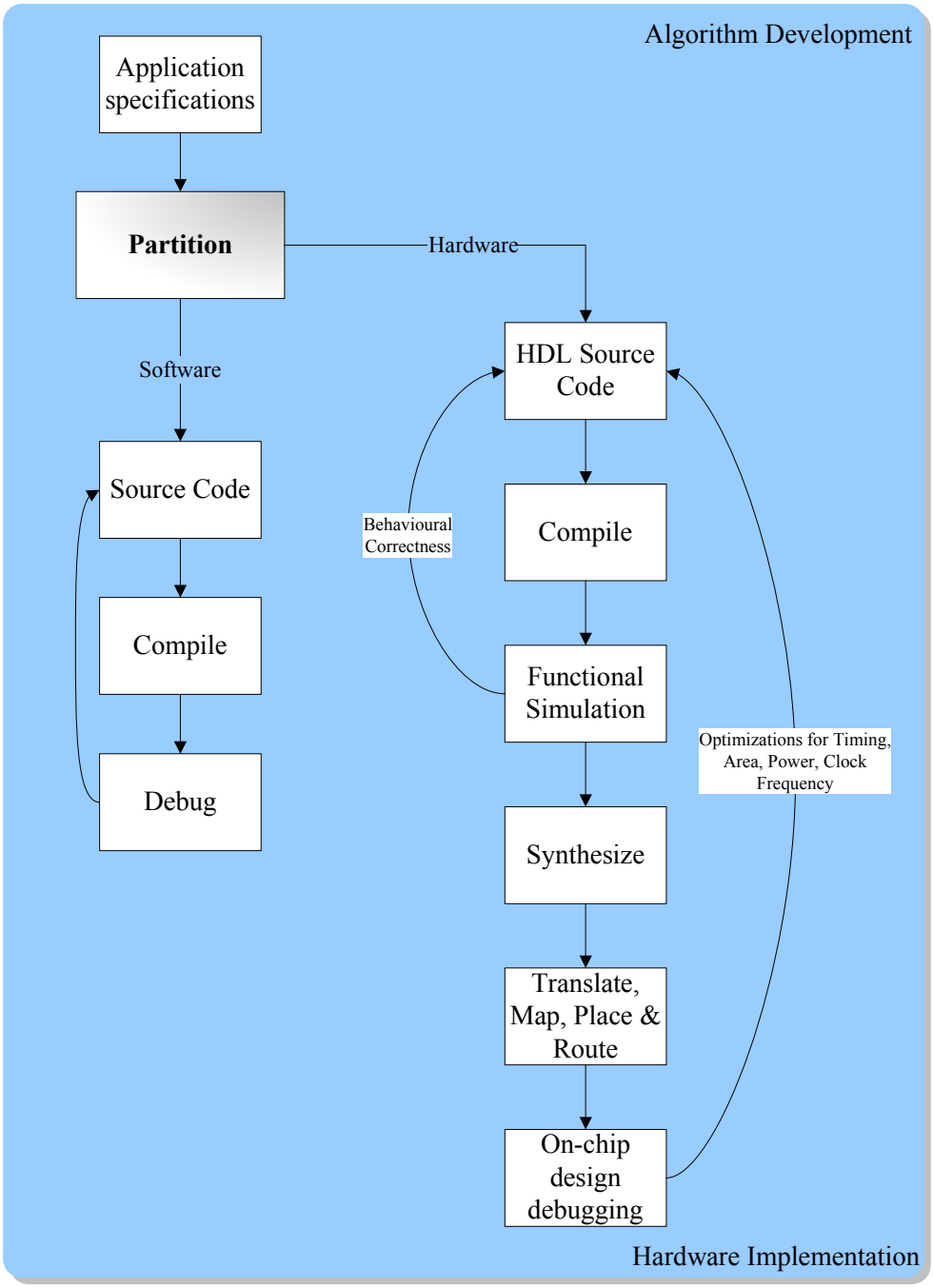


Figure 7.2. Data Flow for RC System with Algorithm Partition (Gokhale & Graham, 2005)

Such systems performing algorithm partitioning on-the-fly can be considered as a next step evolution of today's C-to-HDL compilers. One example is the Garp C compiler (Callahan, Hauser, & Wawrzynek, 2000). During compilation it evaluates candidates for execution in the reconfigurable hardware. Then it removes operations unsupported on FPGAs, which will be executed on a CPU. The compiler than trims the paths so that they fit into the FPGA and breaks down the long ones to increase

performance (automatic introducing of stage delays as in 5.3.2). Finally, Garp estimates hardware versus software execution of the candidate loop and decides on its implementation: reconfigurable hardware or CPU.

Many reference designs for contemporary FPGA-featured boards have entirely demonstrational capabilities with fixed functionality. Such designs have little or no service for the actual end-users who might benefit from using the boards. As demonstrated, the Stratix II GX PCI Express Development Kit and similar products equipped with high-speed communicational interfaces featuring reconfigurable hardware, can be efficiently employed in CPU+FPGA architecture and perform computationally-intensive applications of the DSP field. In reality, a DSP system developer first has to modify and debug existing communicational design(s) or whatever I/O framework is available to her or him. Often, this wastes a significant amount of development time and ends up in creating the required I/O interfaces completely from scratch. Instead, the DSP research community can start contributing to the shared pool of resources – an open-source repository of modified, re-worked and amended designs along with contributing entirely new implementations of various functionalities (I/O interfaces, memory controllers and drivers, etc). Such a repository or library might help the research community to avoid wastage of development time on re-developing the same functionalities all over again and elaborate on the actual designs' algorithms, rather than be obstructed with implementation technicalities and difficulties of I/O interfaces, memory operations, etc. In the case of this project, the contribution for the Stratix II GX PCI Express Development Kit for this library can offer a clear-text HDL design supporting DMA with well-established, tested and verified data exchange via PCIe link. The developed I/O framework can be a contribution to this repository if it is ever established.

References

Agarwal, R., B.V.R.Reddy, & K.K.Aggarwal. (2006). A Switching Mechanism Detection to Reduce Complexity in Multiuser Detection for DS-CDMA Systems. *Journal of Mathematics and Statistics, 2*(2), 368-372.

Altera Corporation (2002). FPGAs Provide Reconfigurable DSP Solutions. Retrieved from http://www.altera.com/literature/wp/wp_dsp_fpga.pdf

Altera Corporation. (2006). *PCI Express to DDR2 SDRAM Reference Design*. Retrieved from http://www.altera.com/literature/an/an431.pdf

Altera Corporation. (2007a). Design Debugging Using the SignalTap II Embedded Logic Analyzer. In *Quartus II Version 7.2 Handbook*.

Altera Corporation. (2007b). *Interfacing DDR2 SDRAM with Stratix II, Stratix II GX, and Arria GX Devices (ver. 4.0)*. Retrieved from http://www.altera.com/literature/an/an328.pdf

Altera Corporation. (2007c). *Nios Development Board Cyclone II Edition Reference Manual*. Retrieved June 9, 2008, from http://www.altera.com/literature/manual/mnl-s2gx-pci-express-devkit.pdf

Altera Corporation. (2007d). *Stratix II GX PCI Express Development Board Reference Manual*. Retrieved June 9, 2008, from http://www.altera.com/literature/manual/mnl-s2gx-pci-express-devkit.pdf

Altera Corporation. (2007e). Welcome to the Quartus II Software. In *Quartus II Help Version 7.2*.

Altium Limited. (2008). *Altium Designer*. Retrieved August 13, 2008, from http://www.altium.com/products/altiumdesigner/

Andrews, D., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbie, M., et al. (2004). Programming models for hybrid FPGA-CPU computational components: a missing link. *IEEE Micro, 24*(4), 42-53.

Baran, P., Bodenner, R., & Hanson, J. (2004). Reduce Build Costs by Offloading DSP Functions to an FPGA. *FPGA and Structured ASIC*.

Beckman, P. (2008). Looking toward Exascale Computing. On *The Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08)* [Keynote Speaker]. Dunedin: University of Otago.

Betz, V., Rose, J., & Marquardt, A. (1999). *Architecture and CAD for Deep-Submicron FPGAs*. Norwell: Kluwer Academic Publishers.

Buell, D., El-Ghazawi, T., Gaj, K., & Kindratenko, V. (2007). Guest Editors' Introduction: High-Performance Reconfigurable Computing. *Computer, 40*(3), 23-27.

Callahan, T. J., Hauser, J. R., & Wawrzynek, J. (2000). The Garp architecture and C compiler. *Computer, 33*(4), 62-69.

Carroll, B. W., & Ostlie, D. A. (2007). 6.3 Radio Telescopes. In *An introduction to modern astrophysics* (2nd ed.). San Francisco: Pearson Addison-Wesley.

Chang, B.-J. (2007). Markov Decision Process Based Multiple Codes Assignment in UMTS WCDMA Mobile Networks. *Wireless Personal Communications, 41*(3), 325-344.

Chang, C. (2005). *Design and Applications of a Reconfigurable Computing System for High Performance Digital Signal Processing*. Unpublished PhD thesis, University of California, Berkeley. Retrieved from http://bwrc.eecs.berkeley.edu/Publications/2005/THESES/c.chang/ChenChang_PhD_thesis.pdf

Chang, C., Wawrzynek, J., & Brodersen, R. W. (2005). BEE2: a high-end reconfigurable computing system. *Design & Test of Computers, IEEE, 22*(2), 114-125.

Deller, A. (2005). *An FX software correlator for VLBI*. Retrieved March 28, 2008, from http://www.atnf.csiro.au/vlbi/evlbi2005/presentations/evlbi_Workshop2/Deller-eVLBI.ppt

Deller, A. B T., Tingay, S. B J., Bailes, M., & West, C. (2007). DiFX: A Software Correlator for Very Long Baseline Interferometry Using Multiprocessor Computing Environments. *Publications of the Astronomical Society of the Pacific, 119*(853), 318-336.

Ekas, P. (2007). FPGAs rapidly replacing high-performance DSP capability. *DSP-FPGA.com*. Retrieved from http://www.dsp-fpga.com/articles/ekas/

Elmore, J. G., Barton, M. B., Moceri, V. M., Polk, S., Arena, P. J., & Fletcher, S. W. (1998). Ten-Year Risk of False Positive Screening Mammograms and Clinical Breast Examinations. *N Engl J Med, 338*(16), 1089-1096.

Escoffier, R. (1997). *The MMA Correlator*. Soccoro, New Mexico: National Radio Astronomy Observatory. Retrieved from http://www.alma.nrao.edu/memos/html-memos/abstracts/abs166.html

Estrin, G. (1960). *Organization of Computer Systems - The Fixed Plus Variable Structure Computer*. Paper presented at the Western Joint Computer Conference, New York. from doi:10.1145/321105.321110

Estrin, G. (2002, October-December). Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. *IEEE Annals of the History of Computing, 24,* 3-9.

Ferris, D. (2006). *The CABB Correlator*. Paper presented at the Next Generation Correlators for Radio Astronomy and Geodesy. Retrieved 25 May, 2008, from http://www.radionet.eu/rda/archive/NA4-EN-SU-009-022_Ferris2.ppt

Fingeroff, M., Gardner, D., & Hogan, M. (2007). Top-down DSP design for FPGAs. *Programmable Logic DesignLine*.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers, 21*(9).

Galanis, M. D., Milidonis, A., Theodoridis, G., Soudris, D., & Goutis, C. E. (2007). Automated framework for partitioning DSP applications in hybrid reconfigurable platforms. *Microprocessors and Microsystems, 31*(1), 1-14.

Ganousis, D. (2004). Top-Down DSP Design Flow to Silicon Implementation. *FPGA and Structured ASIC Journal*. Retrieved from http://www.fpgajournal.com/articles/accelchip_topdown.htm

Gentile, A., & Wills, D. S. (2004). Portable video supercomputing. *Computers, IEEE Transactions on, 53*(8), 960-973.

Gokhale, M., Frigo, J., Ahrens, C., Popkin-Paine, M., & Stone, J. M. (2004). Streams-C; Sc2 C-to-FPGA compiler. *STAR, 42*(5).

Gokhale, M., & Graham, P. S. (2005). *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*: Springer.

Guccione, S. A. (2008). Recongigurable Computing Systems. In S. Hauck & A. DeHon (Eds.), *Reconfigurable computing: the theory and practice of FPGA-based computation* (pp. 908). Amsterdam ; Boston: Morgan Kaufmann.

Guo, Z., Najjar, W., Vahid, F., & Vissers, K. (2004, 22-24 February). *A quantitative analysis of the speedup factors of FPGAs over processors*. Paper presented at the ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA, Monterey, CA. from http://www.scopus.com/scopus/inward/record.url?eid=2-s2.0-2442575888&partner=40&rel=R5.0.4

Hoff, M., Jr., & Townsend, M. (1979). An analog input/output microprocessor for signal processing. *Solid-State Circuits Conference. Digest of Technical Papers. 1979 IEEE International, XXII*, 220-221.

Hutchings, B. L., & Nelson, B. E. (2008). Implememnting Applications with FPGAs. In S. Hauck & A. DeHon (Eds.), *Reconfigurable computing: the theory and practice of FPGA-based computation* (pp. 908). Amsterdam ; Boston: Morgan Kaufmann.

Huynh, P. T., Jarolimek, A. M., & Daye, S. (1998). The false-negative mammogram. *Radiographics, 18*(5), 1137-1154.

IBM Corporation. (2006). *Blue Gene specification sheet*, 2006). Retrieved from http://www-03.ibm.com/systems/resources/systems_deepcomputing_pdf_bluegene_spec_sheet.pdf

Ibnkahla, M. (2004). *Signal Processing for Mobile Communications Handbook*: CRC Press.

Imai, H., Koyama, Y., & Kondo, T. (2005). *K5 software correlator user manual (revised version)*. from http://www2.nict.go.jp/w/w114/stsi/K5/VSSP/K5corr_users_manual.pdf

Impulse Accelerated Technologies. (2008a). CoBuilder C Pragmas. In *CoDeveloper User Guide*.

Impulse Accelerated Technologies. (2008b). *CoDeveloper from Impulse Accelerated Technologies. XtremeData XD1000 Platform Support Package*, 2008b).

Impulse Accelerated Technologies. (2008c). Constraints for Hardware Processes. In *CoDeveloper User Guide*.

Impulse Accelerated Technologies. (2008d). The Programming Model. In *CoDeveloper User Guide*.

Impulse Accelerated Technologies. (2008e). Understanding Loop Pipelining. In *CoDeveloper User Guide*.

Jungo Ltd. (2008). *Driver Development for USB/PCI: WinDriver*. Retrieved May 27, 2008, from http://www.jungo.com/st/windriver.html

Kawaguchi, N., Kobayashi, H., & Oyama, T. (2006). *New Correlator Developments in Japan*. Retrieved May 25, 2008, from http://www.radionet.eu/rda/archive/NA4-EN-SU-009-016_New%20Correlator%20Developments%20in%20Japan.pdf

Kenny, R. (2007, December). FPGA signal processing for radar/sonar applications. *RF Design*. Retrieved from http://rfdesign.com/military_defense_electronics/712DEF2.pdf

Kitaev, S., & Molteno, T. (2008). How to Compute Faster and Cheaper: Reconfigurable HPC. *PDCAT'08*.

Kuo, S. M., & Lee, B. H. (2001). *Real-Time Digital Signal Processing: Implementations, Applications, and Experiments with the Tms320c55x*: John Wiley & Sons, Ltd.

Leonov, M., & Kitaev, V. (2007). *Multi-Channel Correlation of Wideband RF Signals In Hybrid CPU+FPGA System.* Paper presented at the The 14th Electronics New Zealand Conference (ENZCon), Wellington, New Zealand. from

Leonov, M., & Kitaev, V. (2008). *Feasibility Study of Implementing Multi-Channel Correlation for DSP applications on Reconfigurable CPU+FPGA Platform.* Paper presented at the Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08), Dunedin, New Zealand. from

Li, X., Bond, E. J., Veen, B. D. V., & Hagness, S. C. (2005). An overview of ultra-wideband microwave imaging via space-time beamforming for early-stage breast-cancer detection. *Antennas and Propagation Magazine, IEEE, 47*(1), 19-34.

Lowe, S. T. (2004). *Softc: an Operational Software Correlator.* Paper presented at the IVS 2004 General Meeting Proceedings, Ottawa, Canada. from http://ivs.nict.go.jp/mirror/publications/gm2004/

Maxim Integrated Products, Dallas Semiconductor. (2001). *MAX105EVKIT Evaluation Kit for the MAX105, MAX107*. Retrieved May 28, 2008, from http://www.maxim-ic.com/quick_view2.cfm/qv_pk/3119

Meyer-Baese, U. (2004a). *Digital signal processing with field programmable gate arrays* (Second ed.): Springer Berlin Heidelberg.

Meyer-Baese, U. (2004b). Introduction. In *Digital signal processing with field programmable gate arrays* (Second ed.): Springer Berlin Heidelberg.

Milrod, J. (2006). The future of high-performance COTS signal processing Hybrid FPGA/DSP architecture: the optimal solution. *DSP-FPGA.com Resource Guide*.

Mitra, S. K. (2006). *Digital Signal Processing: A Computer-Based Approach* (3rd ed.). New York: McGraw-Hill.

Mlynek, D. (1999). *Design of VLSI Systems. Chapter 12 - Digital Signal Processing Architectures. History*. Retrieved May 25, 2008, from http://lsiwww.epfl.ch/LSI2001/teaching/webcourse/ch12/DSParch.htm#12.2

Nicholson, W. E., Blasco, R. W., & Reddy, K. R. (1978). S2811 Signal Processing Peripheral *Wescon Technical Papers, 22*, 12.

PCI Special Interest Group. (1999). *PCI-X 1.0 Protocol Specification*.

Pellerin, D., & Thibault, S. (2005). 6.6. Hardware Generation Notes. In *Practical FPGA Programming in C* (pp. 464): Prentice Hall Press.

Phillips, S., Littlefield, M., Dahlgren, K., & Ciufo, C. (2007, October 25, 2007). *FPGAs and Reconfigurable Computing replace DSPs*. from https://event.on24.com/eventRegistration/EventLobbyServlet?target=registration.jsp&eventid=95580

Romein, J. W., Broekema, P. C., Meijeren, E. v., Schaaf, K. v. d., & Zwart, W. H. (2006a). *Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer*. Paper presented at the Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures, Cambridge, Massachusetts, USA. from doi:http://doi.acm.org/10.1145/1148109.1148118

Romein, J. W., Broekema, P. C., Meijeren, E. v., Schaaf, K. v. d., & Zwart, W. H. (2006b, June 27-29). *The LOFAR Blue Gene/L Correlator*. Paper presented at the Next Generation Correlators for Radio Astronomy and Geodesy, Groningen, The Netherlands. from www.radionet-eu.org/rda/archive/NA4-EN-SU-009-033_Romein.ppt

S.Bhaktavatsala. (2002). *DSP applications in radar*. Bombay: Indian Institute of Technology.

Stergiopoulos, S. (2000). *Advanced signal processing handbook [electronic resource] : theory and implementation for radar, sonar, and medical imaging real time systems* (Vol. 0849336910).

Tahernia, O. (2005). Reconfigurable DSPs: they're fast and flexible, but are they accessible? *DSP-FPGA.com Product Resource Guide*.

Telikepalli, A., & Fiset, E. (2006). Platform FPGA design for high-performance DSPs. *embedded.com*. Retrieved from http://www.embedded.com/columns/showArticle.jhtml?articleID=185302501

Texas Instruments Inc. (2008). *TMS320C6474 Multicore Digital Signal Processor*.

Thompson, A. R., Moran, J. M., & Swenson, G. W. (2001a). 8.7 Digital Correlators. In *Interferometry and Synthesis in Radio Astronomy* (2nd ed., pp. 283-298). Berlin: Wiley-VCH.

Thompson, A. R., Moran, J. M., & Swenson, G. W. (2001b). *Interferometry and Synthesis in Radio Astronomy* (2nd ed.). Berlin: Wiley-VCH.

Tripp, J. L., Peterson, K. D., Ahrens, C., Poznanovic, J. D., & Gokhale, M. B. (2005). *Trident: an FPGA compiler framework for floating-point algorithms*. Paper presented at the International Conference on Field Programmable Logic and Applications. from doi:10.1109/FPL.2005.1515741

Turney, R. D., Dick, C., Parlour, D. B., & Hwang, J. (2000). Modelling and implementation of DSP FPGA solutions. Retrieved from http://www.xilinx.com/products/logicore/dsp/matlab_final.pdf

Wain, R., Bush, I., Guest, M., Deegan, M., Kozin, I., & Kitchen, C. (2006). *An overview of FPGAs and FPGA programming; Initial experiences at Daresbury*. Daresbury, Warrington, Cheshire, WA4 4AD, UK: Computational Science and Engineering Department, CCLRC Daresbury Laboratory.

West, C. J. (2004). *Development of disk-based baseband recorders and software correlators for radio astronomy*. Swinburne University Of Technology, Swinburne.

Wilkinson, B., & Allen, M. (2004). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers* (2nd ed.). New Jersey: Prentice Hall.

www.top500.org. Retrieved March 03, 2008, from http://www.top500.org/

Xilinx Inc. *Electronic System Level Design Ecosystem*. Retrieved February 21, 2008, from http://www.xilinx.com/products/design_tools/logic_design/advanced/esl/

Xilinx Inc. (1984). *Our History*. Retrieved May 29, 2008, from http://www.xilinx.com/company/history.htm#begin

XtremeData, Inc. *XD1000™ Development System. XD1000™ FPGA Coprocessor Module. User PC Setup Procedure*. Retrieved March 03, 2008, from http://www.xtremedatainc.com/

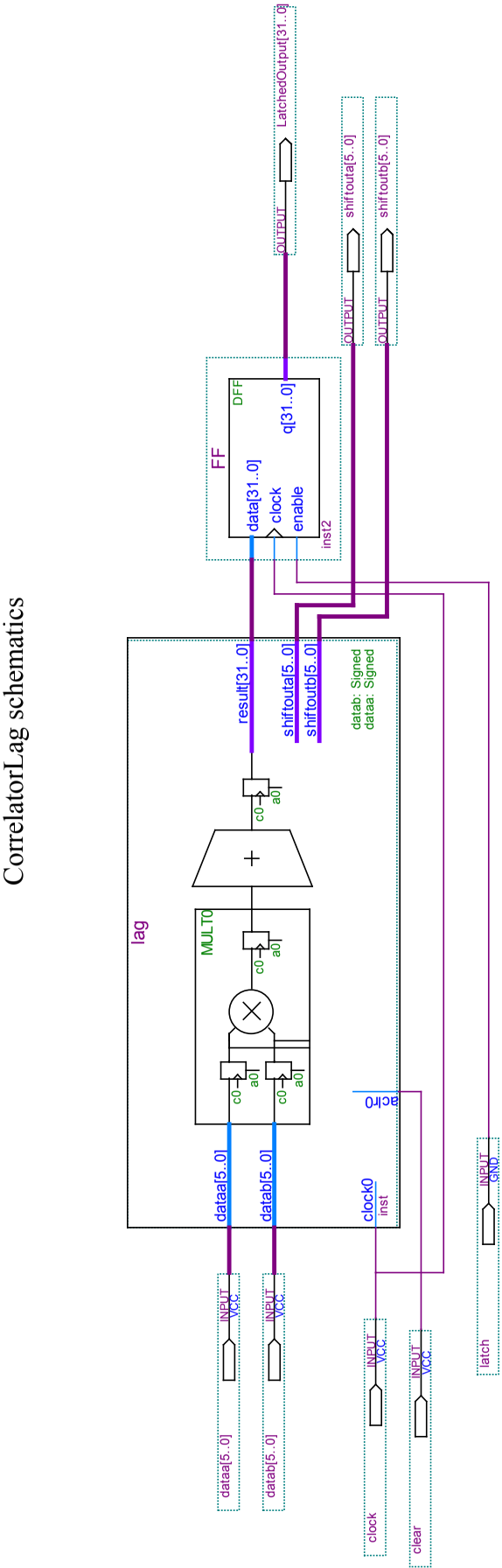
Appendix A1. Components of the 32-lag Hardware Correlator Design

Correlator.vhd (top-level entity).

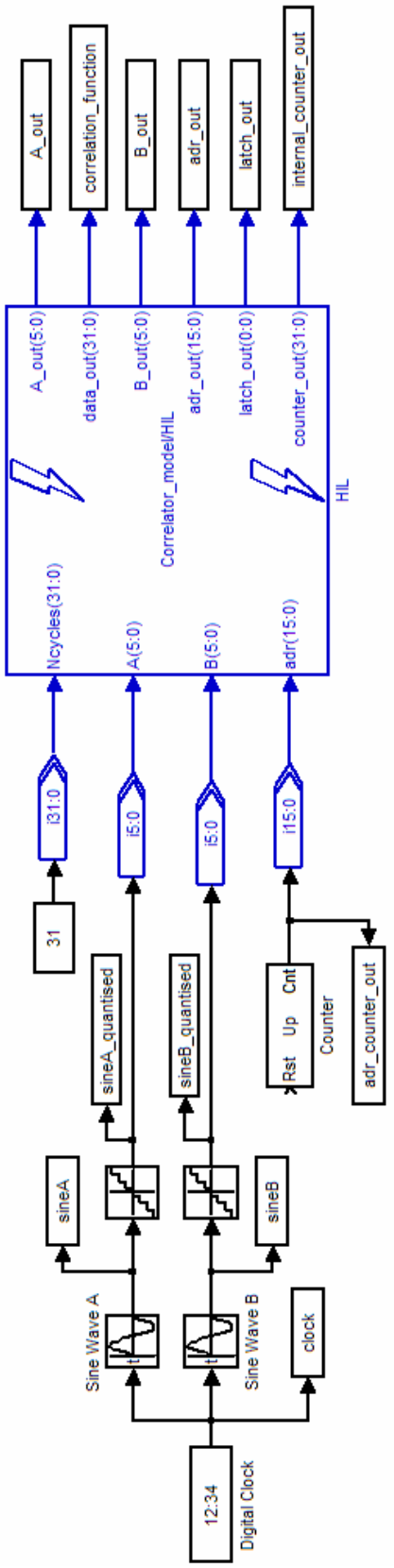
```
--Code was created by Maxim Leonov
--Copyright 2007 Auckland University of Technology
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Correlator is
   generic(LAGS : integer := 32);
   port (
                    clk          : in   std_logic;
                    A            : in   std_logic_vector(5 downto 0);
                    B            : in   std_logic_vector(5 downto 0);

                    Ncycles      : in   unsigned (31 downto 0);
                    adr    : in    unsigned(15 downto 0);
                    adr_out      : out  unsigned(15 downto 0);
                    counter_out  : out  unsigned(31 downto 0);
                    data_out     : out  std_logic_vector(31 downto 0)
       );
end entity;

architecture C_logic of Correlator is
   component CorrelatorLag is
   port (
            clock        :      IN  std_logic;
            latch        :      IN  std_logic;
            dataa        :      IN    std_logic_vector(5 downto 0);
            datab        :      IN    std_logic_vector(5 downto 0);
            clear        :      IN    std_logic;
            overflow     :      OUT  std_logic;
            LatchedOutput :  OUT  std_logic_vector(31 downto 0);
            shiftouta :  OUT  std_logic_vector(5 downto 0);
            shiftoutb :  OUT  std_logic_vector(5 downto 0)
       );
   end component;

type INTERCONNECT is array (0 to LAGS) of std_logic_vector(5 downto 0);
      signal Intercon : INTERCONNECT;

type RESULTS is array (0 to LAGS - 1) of std_logic_vector(31 downto 0);
      signal LatchedResults : RESULTS;

      type CLRACCUMS is array (0 to LAGS - 1) of std_logic;

signal ClearAccums  : CLRACCUMS;
      signal latch        : std_logic;
      signal LastLatch          : std_logic;
      signal LastLastLatch      : std_logic;
      signal OverflowLatch      : std_logic;
      signal SynchronousLatch   : std_logic;
      signal counter            : unsigned (31 downto 0);


begin

   -- Generate all the lag elements and interconnect
   ELEMENTS:for LAG in 0 to LAGS-1 generate
            lagelement: CorrelatorLag port map(clk,                    --
clock
```

```
      SynchronousLatch,   --latch

      Intercon(LAG),      --dataa

      B,                  --datab

      ClearAccums(LAG),   --clear

      OverflowLatch,      --overflow

      LatchedResults(LAG),--LatchedOutput

      Intercon(LAG+1)     --shiftoutA

      );
   end generate ELEMENTS;

    -- Pass the "A" data to the first lag
      Intercon(0) <= A;


 process (clk, adr, counter)
   begin

      if (rising_edge(clk)) then

          if (counter = Ncycles) then
             latch <= '1';
             ClearAccums(TO_INTEGER(adr)) <= '1';
             counter <= "00000000000000000000000000000000";

          else latch <= '0';
             counter <= counter + 1; -- counting clock cycles
             ClearAccums(TO_INTEGER(adr)) <= '0';
          end if;

      -- drive data to output port according to address requested
             LastLatch <= latch;
             LastLastLatch <= LastLatch;
             SynchronousLatch <= LastLatch and not LastLastLatch;
             -- SynchronousLatch goes high for one clock cycle
             data_out <= LatchedResults(TO_INTEGER(adr));


      end if;
      counter_out <= counter;
      adr_out <= adr;

end process;

end architecture;
```

CorrelatorLag schematics



Appendix A2. Simulink Test Model for 32-lag Hardware Correlator Design



Appendix A3. MATLAB Script to Generate Model Signals for Correlation

sine_generation.m

```
%%Code was created by Maxim Leonov
%%Copyright 2007 Auckland University of Technology
N = 4194304;
A = 9;
F = 32*10^6;
Fs = 256*10^6;
t = 0:1/Fs:N/Fs;
x = round(awgn(A*cos(2*pi*t*F), 3, 'measured', [1]))';
fid = fopen('input_signals.txt', 'w+');
fprintf(fid, '%i\n', x);
fclose(fid);
```

Appendix A4. Reference Software N-Channel Correlation Program

correlation.c

```
//Code was created by Maxim Leonov
//Copyright 2008 Auckland University of Technology
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <windows.h>
#include <winbase.h>
#define length 64
#define Nlag 32      //number of correlator's lags
#define Ns 6        //number of signals or CHANNELS
#define INPUT_FILE_A "input_signals.txt"
#define sequences_length  4194304//input sequence length 4096

main()
{
       int i,j,k,p,lag;
       int count_x = 0;
       int end_x = 0;
       int out_count;
       char c;
       float signals[Ns][length];
       float results[(Ns*(Ns-1))/2][Nlag];
       float *file1 = (float*)malloc(sequences_length * sizeof *file1);
       LARGE_INTEGER ticksPerSecond;
       LARGE_INTEGER tick;   // A point in time
       LARGE_INTEGER start_ticks, end_ticks, cputime;

       FILE *result_file;
       FILE *f1;

       result_file = fopen("results.txt","w");

       for (i = 0; i < (Ns*(Ns-1))/2; i++ )
       {
              for(j = 0; j < Nlag; j++)
              {
                     results[i][j]=0;
              }
       }

       f1 = fopen(INPUT_FILE_A, "r");
       if ( f1 == NULL ) {
              fprintf(stderr, "Error opening input file %s\n",
              INPUT_FILE_A);
              c = getc(stdin);
              exit(-1);
       }

       for(i=0; i<sequences_length; i++)
       {
              fscanf(f1, "%f", &file1[i]);
       }

/***   start time measurement     */
printf ("Value of CLOCKS_PER_SEC is :    %i  ticks/sec\n",CLOCKS_PER_SEC   );
// get the high resolution counter's accuracy
if (!QueryPerformanceFrequency(&ticksPerSecond))
       printf("\tno go QueryPerformance not present");
printf ("\tfreq test:   %I64Ld ticks/sec\n",ticksPerSecond    );
```

```
// what time is it?
if (!QueryPerformanceCounter(&tick) )
       printf("no go counter not installed");
printf ("QueryPerformanceCounter testpoint :   %I64Ld  ticks\n",tick);
QueryPerformanceCounter(&start_ticks);

       /*Start of Correlation*/
do
{
       for ( i = count_x, j = 0; i < count_x+length; i++, j++ ) //acquiring
       length samples from input sequence
       {
              if (i >= sequences_length)
              {
                     end_x = 1;
                     break;
              }
              for(p = 0; p < Ns; p++)
              signals[p][j] = file1[i];
       }
       if (end_x == 0)//processed the whole input sequence?
       {
              out_count = 0;
              for(p = 0; p < Ns-1; p++)//selecting first signal
              {
                     for(k = p+1; k < Ns; k++)   //selecting second signal
                     {
                            for (lag = 0; lag < Nlag; lag++) //running through
                            lags
                            {
                                   //running through signal values
                                   for(j=length/2; j < length; j++)
                                   {
results[out_count][lag] += signals[k][j]*signals[p][j-lag];
                                   }
                            }
                            out_count++;
                     }
              }
              //moving across input sequence using length window
              count_x = count_x + length;
       }
} while(end_x == 0);
       /*End of Correlation*/

QueryPerformanceCounter(&end_ticks);
cputime.QuadPart = end_ticks.QuadPart- start_ticks.QuadPart;

printf ("\tElapsed CPU time test:   %.9f  sec  ticks %d\n",
       ((float)cputime.QuadPart/(float)ticksPerSecond.QuadPart),
       cputime.QuadPart);

/***   end time measurement       */

//print results into file
       for (i = 0; i < (Ns*(Ns-1))/2; i++ )
       {
              fprintf(result_file,"%i: ", i);
              for(j = 0; j < Nlag; j++)
              {
                     fprintf(result_file,"%f\n", results[i][j]);
              }
              fprintf(result_file,"\n\n");
       }

       fclose(result_file);
}
```

Appendix A5. 6-Channel Correlator Impulse CoDeveloper project

Correlator_C_sw.c

```
////////////////////////////////////////////////////////////////////////////////
/
//
// Generated by Impulse CoDeveloper
// Impulse C is Copyright(c) 2003-2007 Impulse Accelerated Technologies, Inc.
//
// Correlator_C_sw.c: includes the software test bench processes and
// main() function.
// Code was modified by Maxim Leonov
// Copyright 2008 Auckland University of Technology
//

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "Correlator_C.h"
// #include <sys/time.h>
// #include <time.h>
// #include <unistd.h>
#include <malloc.h>

INTYPE *filebuffer;
const char * FileName = OUTPUT_FILE;
FILE * outFile;

extern co_architecture co_initialize(void *);

void Producer(co_stream x1, co_stream x2, co_stream x3, co_stream x4,
co_stream x5, co_stream x6)
{

       int j, i;
       //struct timeval t1;
       //struct tm *pt1;
       char t1_str[40];

       IF_SIM(cosim_logwindow log = cosim_logwindow_create("Producer");)

       co_stream_open(x1, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
       co_stream_open(x2, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
       co_stream_open(x3, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
       co_stream_open(x4, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
       co_stream_open(x5, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
       co_stream_open(x6, O_WRONLY, INT_TYPE(INSTREAMWIDTH));

       IF_SIM(cosim_logwindow_write(log, "Sending test data...\n");)

//     gettimeofday(&t1, NULL);
//     pt1 = localtime(&t1.tv_sec);
//     strftime(t1_str, sizeof(t1_str), "%Y-%m-%d %H:%M:%S", pt1);
//     printf("Start: %s.%ld (%ld microseconds) \n", t1_str, t1.tv_usec/1000,
t1.tv_usec);

       for(i=0; i<SEQUENCES_LENGTH;i++) {
              co_stream_write(x1, &filebuffer[i], sizeof(INTYPE));
              co_stream_write(x2, &filebuffer[i], sizeof(INTYPE));
              co_stream_write(x3, &filebuffer[i], sizeof(INTYPE));
              co_stream_write(x4, &filebuffer[i], sizeof(INTYPE));
              co_stream_write(x5, &filebuffer[i], sizeof(INTYPE));
```

```
            co_stream_write(x6, &filebuffer[i], sizeof(INTYPE));
            IF_SIM(cosim_logwindow_fwrite(log, "i=%d Value: 0x%x\n", i,
filebuffer[i]);)
      }

      IF_SIM(cosim_logwindow_write(log, "Finished writing test data.\n");)
      co_stream_close(x1);
      co_stream_close(x2);
      co_stream_close(x3);
      co_stream_close(x4);
      co_stream_close(x5);
      co_stream_close(x6);

}

void Consumer(co_stream outstr)
{

      OUTTYPE testValue;
      unsigned int count = 0;


      OUTTYPE k;
      OUTTYPE buffer[NLAG];
      //struct timeval t1;
      //struct tm *pt1;
      char t1_str[40];
      int i = 0;

IF_SIM(cosim_logwindow log = cosim_logwindow_create("Consumer");)

co_stream_open(outstr, O_RDONLY, INT_TYPE(OUTSTREAMWIDTH));

IF_SIM(cosim_logwindow_write(log, "Consumer reading data...\n");)

for(i=0; i<NLAG; i++)
{
      co_stream_read(outstr, &buffer[i], sizeof(OUTTYPE));
      IF_SIM(cosim_logwindow_fwrite(log, "Value: 0x%08x\n",
      buffer[i]);)
}

//    gettimeofday(&t1, NULL);
//    pt1 = localtime(&t1.tv_sec);
//    strftime(t1_str, sizeof(t1_str), "%Y-%m-%d %H:%M:%S", pt1);
//    printf("Finish: %s.%ld (%ld microseconds) \n", t1_str,
      t1.tv_usec/1000, t1.tv_usec);

for(i=0; i<NLAG; i++)
{
      printf("Filtered value %d: %d\n", i, buffer[i]);
      fprintf(outFile, "%d\n", buffer[i]);
      //IF_SIM(cosim_logwindow_fwrite(log, "Value: 0x%08x\n", buffer[i]);)

      count++;
}

      //printf("\n\n");
      fprintf(outFile, "\n\n");

      IF_SIM(cosim_logwindow_fwrite(log,
            "Consumer read %d filtered data values\n", count);)
      co_stream_close(outstr);

}

int main(int argc, char *argv[])
{
      co_architecture my_arch;
      void *param = NULL;
      int c, i;
```

```
	struct timeval t1, t2;
	struct tm *pt1, *pt2;
	long sec;
	long usec;
	const char * InputFile = INPUT_FILE;
	FILE * inFile;

//	double f;
//	char t1_str[40], t2_str[40];

//	cosim_logwindow_init();
//	cosim_logwindow_create(str);
	printf("Impulse C is Copyright(c) 2003-2007 Impulse Accelerated
		Technologies, Inc.\n");

	filebuffer = (INTYPE*)malloc(sizeof(INTYPE)*SEQUENCES_LENGTH);
	inFile = fopen(InputFile, "r");

	if ( inFile == NULL ) {
		fprintf(stderr, "Error opening input file %s\n",
				InputFile);
		c = getc(stdin);
		exit(-1);
	}
	// Now read and write the data...

	for(i=0; i<SEQUENCES_LENGTH;i++)
	{
		fscanf(inFile, "%d", &filebuffer[i]);
	}
	fclose(inFile);

	outFile = fopen(FileName, "w");
	if ( outFile == NULL ) {
		fprintf(stderr, "Error opening file %s for writing\n",
				FileName);
		exit(-1);
	}


/***	start time mesurement	*/
//	gettimeofday(&t1, NULL);

	my_arch = co_initialize(param);
	co_execute(my_arch);

//	gettimeofday(&t2, NULL);
/***	end time mesurement */

//	pt1 = localtime(&t1.tv_sec);
//	pt2 = localtime(&t2.tv_sec);
	fclose(outFile);
/*	strftime(t1_str, sizeof(t1_str), "%Y-%m-%d %H:%M:%S", pt1);
	strftime(t2_str, sizeof(t2_str), "%Y-%m-%d %H:%M:%S", pt2);
	printf("\n\n %s %ld \n", t1_str, t1.tv_usec/1000);
	printf("\n\n %s %ld \n", t2_str, t2.tv_usec/1000);*/

//	sec = t2.tv_sec - t1.tv_sec;
//	usec = t2.tv_usec - t1.tv_usec;
//	f = usec/1000;
//	printf("\n\n Elapsed time: %ld microseconds\n", usec);

	printf("\n\nApplication complete. Press the Enter key to
			continue.\n");
	c = getc(stdin);

	return(0);

}
```

Correlator_C_hw.c

```
////////////////////////////////////////////////////////////////////////////
/
//
// Generated by Impulse CoDeveloper
// Impulse C is Copyright(c) 2003-2007 Impulse Accelerated Technologies, Inc.
//
// Correlator_C_hw.c: includes the hardware process and configuration
// function.
//
// See additional comments in Correlator_C.h.
// Code was modified by Maxim Leonov
// Copyright 2008 Auckland University of Technology
//

#include "co.h"
#include "cosim_log.h"
#include "Correlator_C.h"

// Software process declarations (see Correlator_C_sw.c)
// extern void Producer(co_stream instr);
extern void Consumer(co_stream outstr);
extern void Producer(co_stream x1, co_stream x2, co_stream x3, co_stream x4,
co_stream x5, co_stream x6);

//
// This is the hardware process.
//
void demux(co_stream in,
                   co_stream out1, co_stream out2, co_stream out3, co_stream
out4, co_stream out5)
{
    INTYPE data;

       IF_SIM(cosim_logwindow log;)
       IF_SIM(log = cosim_logwindow_create("demux");)

    while ( 1 ) {
        co_stream_open(in, O_RDONLY, INT_TYPE(INSTREAMWIDTH));
        co_stream_open(out1, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
        co_stream_open(out2, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
        co_stream_open(out3, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
        co_stream_open(out4, O_WRONLY, INT_TYPE(INSTREAMWIDTH));
        co_stream_open(out5, O_WRONLY, INT_TYPE(INSTREAMWIDTH));

   while ( co_stream_read(in, &data, sizeof(INTYPE)) == co_err_none )
        {
            co_stream_write(out1, &data, sizeof(INTYPE));
            co_stream_write(out2, &data, sizeof(INTYPE));
            co_stream_write(out3, &data, sizeof(INTYPE));
            co_stream_write(out4, &data, sizeof(INTYPE));
            co_stream_write(out5, &data, sizeof(INTYPE));
        }

        co_stream_close(in);
        co_stream_close(out1);
        co_stream_close(out2);
        co_stream_close(out3);
        co_stream_close(out4);
        co_stream_close(out5);
         IF_SIM(break;)    // Only run once for desktop simulation
    }
}

void correlation(co_stream x1, co_stream y1, co_stream r1)
{
       int i, j, lag;
       co_int8 count;
       INTYPE nSample1[LENGTH];
```

```
	INTYPE nSample2[LENGTH];
	INTYPE Sample1, Sample2;
	OUTTYPE Result_temp[NLAG];
	INTYPE nSample1Copy1[LENGTH], nSample1Copy2[LENGTH],
		nSample1Copy3[LENGTH], nSample1Copy4[LENGTH],
		nSample1Copy5[LENGTH], nSample1Copy6[LENGTH],
		nSample1Copy7[LENGTH];
	INTYPE nSample2Copy1[LENGTH], nSample2Copy2[LENGTH],
		nSample2Copy3[LENGTH], nSample2Copy4[LENGTH],
		nSample2Copy5[LENGTH], nSample2Copy6[LENGTH],
		nSample2Copy7[LENGTH];
	OUTTYPE nResult[NLAG], tmp, tmp1;
	OUTTYPE k;

	IF_SIM(int samplesread; int sampleswritten;)

	IF_SIM(cosim_logwindow log;)
	IF_SIM(log = cosim_logwindow_create("correlation");)

	for(i=0; i< NLAG; i++)
		nResult[i]=0;

	k=0;
	i=0;
	count=0;

	do {	// Hardware processes run forever
		IF_SIM(samplesread=0; sampleswritten=0;)

		co_stream_open(x1, O_RDONLY, INT_TYPE(INSTREAMWIDTH));
		co_stream_open(y1, O_RDONLY, INT_TYPE(INSTREAMWIDTH));
		co_stream_open(r1, O_WRONLY, INT_TYPE(OUTSTREAMWIDTH));



	while ( (co_stream_read(x1, &Sample1, sizeof(INTYPE) ) ==
		co_err_none) &&
		(co_stream_read(y1, &Sample2, sizeof(INTYPE) ) ==
		co_err_none))
				{
					nSample1[count] = Sample1;
					nSample2[count] = Sample2;
					nSample1Copy1[count] = Sample1;
					nSample2Copy1[count] = Sample2;
					nSample1Copy2[count] = Sample1;
					nSample2Copy2[count] = Sample2;
					nSample1Copy3[count] = Sample1;
					nSample2Copy3[count] = Sample2;
					nSample1Copy4[count] = Sample1;
					nSample2Copy4[count] = Sample2;
					nSample1Copy5[count] = Sample1;
					nSample2Copy5[count] = Sample2;
					nSample1Copy6[count] = Sample1;
					nSample2Copy6[count] = Sample2;
					nSample1Copy7[count] = Sample1;
					nSample2Copy7[count] = Sample2;
					IF_SIM(samplesread++;)
IF_SIM(cosim_logwindow_fwrite(log, "nSample1[%d]: %d\n", count,
	nSample1[count]);)
IF_SIM(cosim_logwindow_fwrite(log, "nSample2[%d]: %d\n", count,
	nSample2[count]);)
	k++;
	count++;
	if(count == LENGTH)
	{
	for (lag = 0; lag < NLAG; lag++)
	{
		tmp=nResult[lag];
		tmp1 = tmp;
		for(j = LENGTH/2; j < LENGTH-7; j+=8)
		{
```

```
                    #pragma CO PIPELINE
                    #pragma CO set stageDelay 32
                    tmp1= nSample1[j]*nSample2[j-lag];
                    tmp1+= nSample1Copy1[j+1]*nSample2Copy1[j+1-lag];
                    tmp1+= nSample1Copy2[j+2]*nSample2Copy2[j+2-lag];
                    tmp1+= nSample1Copy3[j+3]*nSample2Copy3[j+3-lag];
                    tmp1+= nSample1Copy4[j+4]*nSample2Copy4[j+4-lag];
                    tmp1+= nSample1Copy5[j+5]*nSample2Copy5[j+5-lag];
                    tmp1+= nSample1Copy6[j+6]*nSample2Copy6[j+6-lag];
                    tmp1+= nSample1Copy7[j+7]*nSample2Copy7[j+7-lag];
                    tmp += tmp1;
             }
             nResult[lag]=tmp;
      }
      count=0;
      }

      if(k == SEQUENCES_LENGTH)
      {
      for(i=0; i< NLAG; i++)
      {
             co_stream_write(r1, &nResult[i], sizeof(OUTTYPE));
             IF_SIM(sampleswritten++;)
             IF_SIM(cosim_logwindow_fwrite(log, "nResult[%d]: %d\n", i,
      nResult[i]);)
      }
      k=0;
      for(i=0; i< NLAG; i++)
             nResult[i]=0;
      break;
      }
}

co_stream_close(x1);
co_stream_close(y1);
co_stream_close(r1);
IF_SIM(cosim_logwindow_fwrite(log,
"Closing filter process, samples read: %d, samples written: %d\n",
          samplesread, sampleswritten);)

IF_SIM(break;)      // Only run once for desktop simulation
} while(1);
}

//
// Impulse C configuration function
//

void config_Correlator_C(void *arg)
{
      int i,j,l,n,t;
      int p;
      int k;
      co_stream instream[NINST];
      co_stream interstream[NINTERST];
      co_stream outstream[NOUTST];
      co_process demux_proc[NINST];
      co_process correlation_proc[NOUTST];
      co_process producer_process;
      co_process consumer_process[NOUTST];

      char *xname[] = {               "instream1",
                                                  "instream2",
                                                  "instream3",
                                                  "instream4",
                                                  "instream5",
                                                  "instream6"};
      char *iname[] = {"interstream1", "interstream2", "interstream3",
                    "interstream4", "interstream5", "interstream6",
                    "interstream7", "interstream8", "interstream9",
                    "interstream10", "interstream11", "interstream12",
```

```
                "interstream13", "interstream14", "interstream15",
                "interstream16", "interstream17", "interstream18",
                "interstream19", "interstream20", "interstream21",
                "interstream22", "interstream23", "interstream24",
                "interstream25", "interstream26", "interstream27",
                "interstream28", "interstream29", "interstream30"};
char *dname[] = {                                "demux1",
                                                 "demux2",
                                                 "demux3",
                                                 "demux4",
                                                 "demux5",
                                                 "demux6"};
char *rname[] = {                                "outstream1",
                                                 "outstream2",
                                                 "outstream3",
                                                 "outstream4",
                                                 "outstream5",
                                                 "outstream6",
                                                 "outstream7",
                                                 "outstream8",
                                                 "outstream9",
                                                 "outstream10",
                                                 "outstream11",
                                                 "outstream12",
                                                 "outstream13",
                                                 "outstream14",
                                                 "outstream15"};
char *producername[] = {                "Producer1",
                                                 "Producer2",
                                                 "Producer3",
                                                 "Producer4",
                                                 "Producer5",
                                                 "Producer6"};
char *correlationname[] = {                      "correlation1",
                                                 "correlation2",
                                                 "correlation3",
                                                 "correlation4",
                                                 "correlation5",
                                                 "correlation6",
                                                 "correlation7",
                                                 "correlation8",
                                                 "correlation9",
                                                 "correlation10",
                                                 "correlation11",
                                                 "correlation12",
                                                 "correlation13",
                                                 "correlation14",
                                                 "correlation15"};
char *Consumername[] = {                "Consumer1",
                                                 "Consumer2",
                                                 "Consumer3",
                                                 "Consumer4",
                                                 "Consumer5",
                                                 "Consumer6",
                                                 "Consumer7",
                                                 "Consumer8",
                                                 "Consumer9",
                                                 "Consumer10",
                                                 "Consumer11",
                                                 "Consumer12",
                                                 "Consumer13",
                                                 "Consumer14",
                                                 "Consumer15"};


IF_SIM(cosim_logwindow_init();)


for (i=0; i<NINST; i++) {
      instream[i]=co_stream_create(xname[i],INT_TYPE(INSTREAMWIDTH),
                                   INSTREAMDEPTH); //input streams
```

```
	}

	for (i=0; i<NINTERST; i++) {
//intermediate streams between input and correlator (avoiding unidirectional
connectivity property of streams)
	interstream[i]=co_stream_create(iname[i],INT_TYPE(INSTREAMWIDTH)
					,INTERSTREAMDEPTH);
	}

	for (i=0; i<NOUTST; i++) {
	outstream[i]=co_stream_create(rname[i],INT_TYPE(OUTSTREAMWIDTH),
					OUTSTREAMDEPTH);
	}

	producer_process = co_process_create("Producer",
	(co_function)Producer, 6, instream[0], instream[1], instream[2],
	instream[3], instream[4], instream[5]);

	for (i=0; i<NOUTST; i++) {
consumer_process[i] =
	co_process_create(Consumername[i],(co_function)Consumer, 1,
			outstream[i]);
	}

	for (i=0; i<NINST; i++) {
		demux_proc[i]=co_process_create(dname[i],
				(co_function)demux, 6,
				instream[i], interstream[i],
				interstream[i+6], interstream[i+12],
				interstream[i+18], interstream[i+24]);
	}

correlation_proc[0]=
co_process_create(correlationname[0],(co_function)correlation,3,
				interstream[0],interstream[1],outstream[0]);
correlation_proc[1]=
co_process_create(correlationname[1],(co_function)correlation,3,
				interstream[6],interstream[2],outstream[1]);
correlation_proc[2]=
co_process_create(correlationname[2],(co_function)correlation,3,
				interstream[12],interstream[3],outstream[2]);
correlation_proc[3]=
co_process_create(correlationname[3],(co_function)correlation,3,
				interstream[18],interstream[4],outstream[3]);
correlation_proc[4]=
co_process_create(correlationname[4],(co_function)correlation,3,
				interstream[24],interstream[5],outstream[4]);
correlation_proc[5]=
co_process_create(correlationname[5],(co_function)correlation,3,
				interstream[7],interstream[8],outstream[5]);
correlation_proc[6]=
co_process_create(correlationname[6],(co_function)correlation,3,
				interstream[13],interstream[9],outstream[6]);
correlation_proc[7]=
co_process_create(correlationname[7],(co_function)correlation,3,
			interstream[19],interstream[10],outstream[7]);
correlation_proc[8]=
co_process_create(correlationname[8],(co_function)correlation,3,
			interstream[25],interstream[11],outstream[8]);
correlation_proc[9]=
co_process_create(correlationname[9],(co_function)correlation,3,
			interstream[14],interstream[15],outstream[9]);
correlation_proc[10]=
co_process_create(correlationname[10],(co_function)correlation,3,
			interstream[20],interstream[16],outstream[10]);
correlation_proc[11]=
co_process_create(correlationname[11],(co_function)correlation,3,
			interstream[26],interstream[17],outstream[11]);
correlation_proc[12]=
co_process_create(correlationname[12],(co_function)correlation,3,
			interstream[21],interstream[22],outstream[12]);
```

```
correlation_proc[13]=
co_process_create(correlationname[13],(co_function)correlation,3,
                    interstream[27],interstream[23],outstream[13]);
correlation_proc[14]=
co_process_create(correlationname[14],(co_function)correlation,3,
                    interstream[28],interstream[29],outstream[14]);

       for (i=0; i<NINST; i++)
              co_process_config(demux_proc[i],co_loc,"PE0");
       for (i=0; i<NOUTST; i++)
              co_process_config(correlation_proc[i],co_loc,"PE0");
}

co_architecture co_initialize(int param)
{
       return(co_architecture_create("Correlator_C_arch",
                    "Generic",config_Correlator_C,(void *)param));
}
```

Correlator_C.h

```
////////////////////////////////////////////////////////////////////////////
/
//
// Generated by Impulse CoDeveloper
// Impulse C is Copyright(c) 2003-2006 Impulse Accelerated Technologies, Inc.
// Code was modified by Maxim Leonov
// Copyright 2008 Auckland University of Technology
//

#define INSTREAMDEPTH 128  /* INPUT buffer size for FIFO in hardware */
#define OUTSTREAMDEPTH 64  /* OUTPUT buffer size for FIFO in hardware */
#define INSTREAMWIDTH 8    /* INPUT buffer width for FIFO in hardware */
#define OUTSTREAMWIDTH 32  /* OUTPUT buffer width for FIFO in hardware */
#define INTERSTREAMDEPTH 128     /* INTERMEDIATE buffer size for FIFO in
hardware */
#define INPUT_FILE "input_signals.txt"
#define OUTPUT_FILE "correlator_out.txt"
#define LENGTH 64
#define NLAG 32
#define SEQUENCES_LENGTH 4194304//4096  40960

typedef int8 INTYPE;
typedef int OUTTYPE;

#define NINST 6
#define NOUTST 15
#define NINTERST 30
```

Appendix A6. Files Modified from the Original PCI Express to DDR2 SDRAM Reference Design

tx_ddr_resp.v

```
//-----------------------------------------------------------------------------
-
// Title         : tx_ddr_resp
// Project       : PCIe-to-DDR2 SDRAM Reference Design
//-----------------------------------------------------------------------------
-
// File          : tx_ddr_resp.v
// Author        : Altera Corporation
//-----------------------------------------------------------------------------
-
// Functional Description:
// This module is part of the TX application layer interfacing with the DDR2
// controller
//-----------------------------------------------------------------------------
---
//
// Copyright 2003 Altera Corporation. All rights reserved.  Altera products
are
// protected under numerous U.S. and foreign patents, maskwork rights,
copyrights and
// other intellectual property laws.
// This reference design file, and your use thereof, is subject to and
governed by
// the terms and conditions of the applicable Altera Reference Design License
Agreement.
// By using this reference design file, you indicate your acceptance of such
terms and
// conditions between you and Altera Corporation.  In the event that you do
not agree with
// such terms and conditions, you may not use the reference design file.
Please promptly
// destroy any copies you have made.
//
// This reference design file being provided on an "as-is" basis and as an
accommodation
// and therefore all warranties, representations or guarantees of any kind
// (whether express, implied or statutory) including, without limitation,
warranties of
// merchantability, non-infringement, or fitness for a particular purpose, are
// specifically disclaimed.  By making this reference design file available,
Altera
// expressly does not recommend, suggest or require that this reference design
file be
// used in combination with any other product not provided by Altera
// Code was modified by Maxim Leonov
// Copyright 2008 Auckland University of Technology

// turn off bogus verilog processor warnings
// altera message_off 10034 10035 10036 10037 10230

// synthesis translate_off
`timescale 1ns / 1ps
// synthesis translate_on

module tx_ddr_resp

( input                                 Clk_i,     // Avalon clock
  input                                 Rstn_i,    // Avalon reset
```

```
  // interface to the Rx pending read FIFO
  input                                 RxPndgRdFifoEmpty_i,
  input      [49:0]                     RxPndgRdFifoDato_i,
  output                                RxPndgRdFifoRdReq_o,

  // interface to the Avalon bus
  input                                 TxReadDataValid_i,
  input      [63:0]                     TxReadData_i,
  output     [63:0]                     TxReadData_o,

  // Interface to the Command Fifo
  output     [127:0]                    CmdFifoDatin_o,
  output                                CmdFifoWrReq_o,

  // Interface to the Avalon Tx Control Module
  output                                CmdFifoBusy_o,

  // Interface to DMA Engine
  input      [63:0]                     DmaDstAdr_i,
  input                                 DmaBusy_i,
  input
DmaReg_app_finished_i,

  // cfg signals
  input      [31:0]                     DevCsr_i,
  input      [12:0]                     BusDev_i
);


wire         sm_rd_fifo;
wire         sm_start_resp;
wire         sm_wait_data1;
wire         sm_wait_data2;
wire         sm_send_first;
wire         sm_send_last;
reg          sm_send_last_reg;
wire         sm_send_max;
wire         sm_send_to_4k;
wire [7:0]   bytes_to_RCB;
wire         over_rd_2dw;
wire         over_rd_1dw;
wire [7:0]   cpl_tag;
wire [15:0]  requester_id;
wire [6:0]   rd_addr;
wire [10:0]  rd_dwlen;
wire [3:0]   fbe;
wire [12:0]  remain_bytes;
wire [15:0]  completer_id;
wire         dma_req;
wire [9:0]   wr_dw_len;

reg  [7:0]   txresp_state;
reg  [7:0]   txresp_nxt_state;
reg          first_cpl_sreg;
reg  [7:0]   bytes_to_RCB_reg;
reg  [13:0]  bytes_to_RCB_add_reg;
reg  [12:0]  curr_bcnt_reg;
reg  [13:0]  curr_bcnt_add_reg;
reg  [12:0]  curr_bcnt_reg_int;
reg  [12:0]  max_payload;
reg  [12:0]  max_payload_reg;
reg  [13:0]  max_payload_add_reg;
reg  [13:0]  payload_cntr;
reg  [13:0]  nxt_payload_cntr;
reg  [13:0]  payload_cntr_stg;
reg  [3:0]   over_rd_bytes;
reg  [3:0]   over_rd_bytes_reg;
reg  [13:0]  over_rd_bytes_add_reg;
reg  [12:0]  bytes_sent;
reg  [12:0]  pkt_size;
reg  [12:0]  pkt_size_reg;
```

```
reg  [12:0]  pkt_size_reg_mod;
reg  [3:0]   coming_send_state;
reg  [3:0]   coming_send_state_reg;
reg  [6:0]   lower_addr;
reg  [6:0]   lower_addr_reg;
reg  [63:0]  wr_addr_reg;
reg          PCIeAddrSpace_i;

wire [1:0]   attr;
wire [12:0]  bytes_to_4KB;
reg  [12:0]  bytes_to_4KB_reg;
reg  [13:0]  bytes_to_4KB_add_reg;
wire [127:0] cpl_header;
wire [127:0] wr_header;
wire [127:0] mem_wr64_header;
wire [127:0] mem_wr32_header;
wire         dma_reg_rd;
wire         c1;
wire         c2;
wire         c3;
wire         c4;
wire         c5;
wire         c6;
reg          c1_reg;
reg          c2_reg;
reg          c3_reg;
reg          c4_reg;
reg          c5_reg;
reg          c6_reg;

reg                TxReadData_reg1;

localparam      TXRESP_RD_FIFO       = 8'h01;
localparam      TXRESP_START_RESP    = 8'h02;
localparam      TXRESP_WAIT_DATA1    = 8'h04;
localparam      TXRESP_WAIT_DATA2    = 8'h08;
localparam      TXRESP_SEND_FIRST    = 8'h10;
localparam      TXRESP_SEND_MAX      = 8'h20;
localparam      TXRESP_SEND_TO_4K    = 8'h40;
localparam      TXRESP_SEND_LAST     = 8'h80;


 /// state machine output assignments

 assign   sm_rd_fifo    = txresp_state[0];
 assign   sm_start_resp = txresp_state[1];
 assign   sm_wait_data1 = txresp_state[2];
 assign   sm_wait_data2 = txresp_state[3];
 assign   sm_send_first = txresp_state[4];
 assign   sm_send_max   = txresp_state[5];
 assign   sm_send_to_4k = txresp_state[6];
 assign   sm_send_last  = txresp_state[7];

always @(posedge Clk_i or negedge Rstn_i)  // state machine registers
  begin
    if(~Rstn_i)
      txresp_state <= TXRESP_RD_FIFO;
    else
      txresp_state <= txresp_nxt_state;
  end

// state machine next state gen

always @(*)
  begin
    case(txresp_state)
      TXRESP_RD_FIFO :
        if(~RxPndgRdFifoEmpty_i)
          txresp_nxt_state = TXRESP_START_RESP;
        else
          txresp_nxt_state = TXRESP_RD_FIFO;
```

```
      TXRESP_START_RESP:  // load byte count reg and calc the first byte 7 bit
of addr
           txresp_nxt_state = TXRESP_WAIT_DATA1;

      TXRESP_WAIT_DATA1:
        txresp_nxt_state = TXRESP_WAIT_DATA2;

      TXRESP_WAIT_DATA2:
       if (payload_cntr >= pkt_size_reg) begin
          case(coming_send_state_reg)
            4'b1000: txresp_nxt_state = TXRESP_SEND_FIRST;
            4'b0100: txresp_nxt_state = TXRESP_SEND_MAX;
            4'b0010: txresp_nxt_state = TXRESP_SEND_TO_4K;
            4'b0001: txresp_nxt_state = TXRESP_SEND_LAST;
            default: txresp_nxt_state = TXRESP_WAIT_DATA2;
         endcase
       end
       else
          txresp_nxt_state = TXRESP_WAIT_DATA2;

      TXRESP_SEND_FIRST:
        txresp_nxt_state = TXRESP_WAIT_DATA1;

      TXRESP_SEND_MAX:
        txresp_nxt_state = TXRESP_WAIT_DATA1;

      TXRESP_SEND_TO_4K:
        txresp_nxt_state = TXRESP_WAIT_DATA1;

      TXRESP_SEND_LAST:
        txresp_nxt_state = TXRESP_RD_FIFO;

      default:
        txresp_nxt_state = TXRESP_RD_FIFO;

    endcase
 end

// decode the max payload size
// constant signal from beginning
always @(DevCsr_i)
  begin
    case(DevCsr_i[14:12])
      3'b000 : max_payload= 128;
      3'b001 : max_payload= 256;
      3'b010 : max_payload= 512;
      3'b011 : max_payload= 1024;
      3'b100 : max_payload= 2048;
      default : max_payload = 2048;
    endcase
  end

always @(posedge Clk_i or negedge Rstn_i)
  begin
     if(~Rstn_i)
       max_payload_reg <= 13'h0000;
     else
       max_payload_reg <= max_payload;
  end

// This signal will only be asserted when sm_rd_fifo
assign RxPndgRdFifoRdReq_o = sm_rd_fifo & ~RxPndgRdFifoEmpty_i;

// These will be available from sm_start_resp till sm_send_last
// mainly constant
assign cpl_tag      = RxPndgRdFifoDato_i[7:0];
assign rd_addr      = RxPndgRdFifoDato_i[14:8];
assign dma_reg_rd   = RxPndgRdFifoDato_i[15];
assign requester_id = RxPndgRdFifoDato_i[31:16];
assign rd_dwlen     = RxPndgRdFifoDato_i[42:32];
```

```
assign fbe          = RxPndgRdFifoDato_i[46:43];
assign dma_req      = RxPndgRdFifoDato_i[47];
assign attr         = RxPndgRdFifoDato_i[49:48];

//assign TxReadData_o = dma_reg_rd ?
{DmaBusy_i,31'h00000000,DmaBusy_i,31'h00000000}
//                                  : TxReadData_i;
assign TxReadData_o = dma_reg_rd ?
{DmaBusy_i,{11{1'b0}},DmaReg_app_finished_i,{19{1'b0}},DmaBusy_i,{11{1'b0}},Dm
aReg_app_finished_i,{19{1'b0}}}
                                  : TxReadData_i;

// Modification
//***********************
always @(posedge Clk_i or negedge Rstn_i)
 begin
    if(~Rstn_i)
      TxReadData_reg1 <= 64'h0000_0000;
    else
               if ( dma_reg_rd )
                       TxReadData_reg1 <=
{{12{1'b0}},DmaReg_app_finished_i,{19{1'b0}},{12{1'b0}},DmaReg_app_finished_i,
{19{1'b0}}} ;
               else
                       TxReadData_reg1 <= TxReadData_i ;
    //else ;
 end
//***********************


// calculate the bytes to RCB that could be 64 or 128Bytes (6 or 7 zeros in
address)
// Available from sm_start_resp
assign bytes_to_RCB = 8'h80 - rd_addr[6:0];

// bytes to 4KB boundary
// Updates in different stages

// pipeline for fmax
always @(posedge Clk_i or negedge Rstn_i) begin
  if(~Rstn_i) begin
    bytes_to_RCB_reg <= 0;
    bytes_to_RCB_add_reg <= 14'h0000;
    max_payload_add_reg <= 14'h0000;
    bytes_to_4KB_add_reg <= 14'h0000;
    curr_bcnt_add_reg <= 14'h0000;
    over_rd_bytes_add_reg <= 14'h0000;
  end
  else begin
    bytes_to_RCB_reg <= bytes_to_RCB;
    bytes_to_RCB_add_reg  <= ~{6'h00,bytes_to_RCB_reg[7:0]} + 14'h0001 +
14'h0008;
    max_payload_add_reg   <= ~{1'b0,max_payload[12:0]} + 14'h0001 + 14'h0008;
    bytes_to_4KB_add_reg  <= ~{1'b0,bytes_to_4KB_reg[12:0]} + 14'h0001 +
14'h0008;
    curr_bcnt_add_reg     <= ~{1'b0,curr_bcnt_reg[12:0]} + 14'h0001 +
14'h0008;
    over_rd_bytes_add_reg <= ~{10'h000,over_rd_bytes_reg[3:0]} + 14'h0001 +
14'h0008;
  end
end

// SR reg to indicate the first completion of a read
always @(posedge Clk_i or negedge Rstn_i)
  begin
     if(~Rstn_i)
       first_cpl_sreg <= 1'b0;
     else if(sm_start_resp)
       first_cpl_sreg <= 1'b1;
     else if(sm_send_first | sm_send_to_4k | sm_send_max | sm_send_last)
       first_cpl_sreg <= 1'b0;
```

```
  end

// special signal for pipelining the calculation of payload_cntr after
// the sm_send_last state
always @(posedge Clk_i or negedge Rstn_i)
  begin
     if(~Rstn_i)
       sm_send_last_reg <= 1'b0;
     else
       sm_send_last_reg <= sm_send_last;
  end

assign c1 = dma_req;
assign c2 = first_cpl_sreg;
assign c3 = (curr_bcnt_reg > bytes_to_RCB_reg);
assign c4 = (curr_bcnt_reg > max_payload_reg);
assign c5 = (curr_bcnt_reg > bytes_to_4KB_reg);
assign c6 = (max_payload_reg > bytes_to_4KB_reg);

/// completion payload counter to keep track of the data byte returned from
avalon
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i) begin
      c1_reg <= 0;
      c2_reg <= 0;
      c3_reg <= 0;
      c4_reg <= 0;
      c5_reg <= 0;
      c6_reg <= 0;
    end
    else begin  // Rstn_i
      c1_reg <= c1;
      c2_reg <= c2;
      c3_reg <= c3;
      c4_reg <= c4;
      c5_reg <= c5;
      c6_reg <= c6;
    end
  end

always @(c1, c2, c3, c4, c5, c6, bytes_to_RCB_reg, curr_bcnt_reg,
max_payload_reg,
         bytes_to_4KB_reg)
  begin
    if ((~c1 && c2 && ~c3) || (~c1 && ~c2 && ~c4) || (c1 && ~c4 && ~c5)) begin
      pkt_size = curr_bcnt_reg;
      coming_send_state = 4'b0001;
    end
    else if ((~c1 && ~c2 && c4) || (c1 && c4 && ~c6)) begin
      pkt_size = max_payload_reg;
      coming_send_state = 4'b0100;
    end
    else if (~c1 && c2 && c3) begin
      pkt_size = bytes_to_RCB_reg;
      coming_send_state = 4'b1000;
    end
    else if (c1 && c5 && c6) begin
      pkt_size = bytes_to_4KB_reg;
      coming_send_state = 4'b0010;
    end
    else begin
      pkt_size = 13'h0000;
      coming_send_state = 4'b0000;
    end
  end

/// completion payload counter to keep track of the data byte returned from
avalon
always @(posedge Clk_i or negedge Rstn_i)
  begin
```

```
    if(~Rstn_i) begin
      pkt_size_reg <= 0;
      coming_send_state_reg <= 0;
    end
    else if (sm_wait_data1) begin  // Rstn_i
      pkt_size_reg <= pkt_size;
      coming_send_state_reg <= coming_send_state;
    end
  end

/// completion payload counter to keep track of the data byte returned from
avalon
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      pkt_size_reg_mod <= 0;
    else if (sm_wait_data2)        // Rstn_i
      // pkt_size_reg_mod <= ~{1'b0,pkt_size_reg[11:0]} + 13'h1002;
      pkt_size_reg_mod <= 13'h1000 - pkt_size_reg[11:0];
  end

// Find out how many bytes are sent in each PCIe packet
always @(sm_send_to_4k, sm_send_first, sm_send_max, sm_send_last,
curr_bcnt_reg,
         max_payload_reg, bytes_to_RCB_reg, bytes_to_4KB_reg)
  begin
    case({sm_send_to_4k, sm_send_first, sm_send_max, sm_send_last})
      4'b0001 : bytes_sent = curr_bcnt_reg;
      4'b0010 : bytes_sent = max_payload_reg;
      4'b0100 : bytes_sent = bytes_to_RCB_reg;
      4'b1000 : bytes_sent = bytes_to_4KB_reg;
      default : bytes_sent = 0;
    endcase
  end

// recode the whole payload_cntr block to solve the timing problem
always @(sm_send_first, sm_send_max, sm_send_to_4k, sm_send_last,
sm_send_last_reg,
        bytes_to_RCB_reg, max_payload_reg, bytes_to_4KB_reg, curr_bcnt_reg,
         over_rd_bytes_reg, TxReadDataValid_i, payload_cntr,
bytes_to_RCB_add_reg,
         max_payload_add_reg, bytes_to_4KB_add_reg, curr_bcnt_add_reg,
         over_rd_bytes_add_reg)
  begin
    case({sm_send_first, sm_send_max, sm_send_to_4k, sm_send_last,
sm_send_last_reg,
         TxReadDataValid_i})
      6'b000001 : nxt_payload_cntr[13:2] = payload_cntr[13:2] + 12'h002;
      6'b100001 : nxt_payload_cntr[13:2] = payload_cntr[13:2] +
bytes_to_RCB_add_reg[13:2];
      6'b010001 : nxt_payload_cntr[13:2] = payload_cntr[13:2] +
max_payload_add_reg[13:2];
      6'b001001 : nxt_payload_cntr[13:2] = payload_cntr[13:2] +
bytes_to_4KB_add_reg[13:2];
      6'b000101 : nxt_payload_cntr[13:2] = payload_cntr[13:2] +
curr_bcnt_add_reg[13:2];
      6'b000011 : nxt_payload_cntr[13:2] = payload_cntr[13:2] +
over_rd_bytes_add_reg[13:2];
      6'b100000 : nxt_payload_cntr[13:2] = payload_cntr[13:2] -
{6'h00,bytes_to_RCB_reg[7:2]};
      6'b010000 : nxt_payload_cntr[13:2] = payload_cntr[13:2] -
{1'b0,max_payload_reg[12:2]};
      6'b001000 : nxt_payload_cntr[13:2] = payload_cntr[13:2] -
{1'b0,bytes_to_4KB_reg[12:2]};
      6'b000100 : nxt_payload_cntr[13:2] = payload_cntr[13:2] -
{1'b0,curr_bcnt_reg[12:2]};
      6'b000010 : nxt_payload_cntr[13:2] = payload_cntr[13:2] -
{10'h000,over_rd_bytes_reg[3:2]};
      default : nxt_payload_cntr[13:2] = payload_cntr[13:2];
    endcase
  end
```

```
// assign nxt_payload_cntr[1:0] = 2'b00;

/// completion payload counter to keep track of the data byte returned from
avalon
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      payload_cntr[13:0] <= 0;
    else // Rstn_i
      payload_cntr[13:0] <= {nxt_payload_cntr[13:2],2'b00};
  end

/// over read bytes caculation due to more data being read from the
// avalon to compensate for the alignment 32/64
// signals ready from start_resp onwards, mainly constant
assign over_rd_2dw = rd_addr[2] & ~rd_dwlen[0];
assign over_rd_1dw = rd_dwlen[0];

always @(over_rd_2dw, over_rd_1dw )
  begin
    case({over_rd_2dw, over_rd_1dw})
      2'b01 : over_rd_bytes = 4;
      2'b10 : over_rd_bytes = 8;
      default: over_rd_bytes = 0;
    endcase
  end

always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      over_rd_bytes_reg <= 0;
    else
      over_rd_bytes_reg <= over_rd_bytes;
  end

// the current byte count register that still need to be sent for PCIe resp
// the remaining byte count after a DMA write header is written into the
Command FIFO
always @(posedge Clk_i or negedge Rstn_i) begin
  if(~Rstn_i)
    curr_bcnt_reg <= 13'h0;
  else if(sm_start_resp)
    curr_bcnt_reg <= {rd_dwlen, 2'b00};
  else if(sm_send_first | sm_send_max | sm_send_to_4k)
    curr_bcnt_reg <= curr_bcnt_reg - pkt_size_reg;
  else if(sm_send_last)
    curr_bcnt_reg <= 0;
  else
    curr_bcnt_reg <= curr_bcnt_reg;
end

// the current byte count register that still need to be sent for PCIe resp
// the remaining byte count after a DMA write header is written into the
Command FIFO
always @(posedge Clk_i or negedge Rstn_i) begin
  if(~Rstn_i)
    curr_bcnt_reg_int <= 13'h0;
  else if(sm_start_resp)
    curr_bcnt_reg_int <= {rd_dwlen, 2'b00} + over_rd_bytes;
  else if(sm_send_first | sm_send_max | sm_send_to_4k)
    curr_bcnt_reg_int <= curr_bcnt_reg_int - pkt_size_reg;
  else if(sm_send_last)
    curr_bcnt_reg_int <= 0;
  else
    curr_bcnt_reg_int <= curr_bcnt_reg_int;
end

/// the remaining bcnt (for the header)
assign remain_bytes = curr_bcnt_reg;
```

```
// read address reg increments byte the amount of byte in each read header
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      wr_addr_reg <= 0;
    else if(sm_start_resp & dma_req & (cpl_tag == 8'h00))
      wr_addr_reg <= DmaDstAdr_i;
    else if(dma_req & (sm_send_max | sm_send_to_4k | sm_send_last))
      wr_addr_reg[15:0] <= wr_addr_reg[15:0] + pkt_size_reg;
    else
      wr_addr_reg[15:0] <= wr_addr_reg[15:0];
  end


// assign bytes_to_4KB = 13'h1000 - wr_addr_reg[11:0];
// read address reg increments byte the amount of byte in each read header
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      bytes_to_4KB_reg <= 0;
    else if(sm_start_resp & dma_req & (cpl_tag == 8'h00))
      bytes_to_4KB_reg <= 13'h1000 - DmaDstAdr_i[11:0];
    else if(dma_req & (sm_send_max | sm_send_to_4k | sm_send_last))
      // bytes_to_4KB_reg <= ~{1'b0,wr_addr_reg[11:0]} +
~{1'b0,pkt_size_reg[11:0]} + 13'h1002;
      // bytes_to_4KB_reg <= ~{1'b0,wr_addr_reg[11:0]} +
pkt_size_reg_mod[12:0];
      bytes_to_4KB_reg <= pkt_size_reg_mod[12:0] - wr_addr_reg[11:0];
    else
      bytes_to_4KB_reg <= bytes_to_4KB_reg;
  end

// ============================================================================
// Gather the info to generate the PCIe Cpl header for PCIe response
// ============================================================================

// calculate the 7 bit lower address of the first enable byte
// based on the first byte enable
always @(fbe,rd_addr)
 begin
  casex(fbe)
    4'bxxx1 : lower_addr = {rd_addr[6:2], 2'b00};
    4'bxx10 : lower_addr = {rd_addr[6:2], 2'b01};
    4'bx100 : lower_addr = {rd_addr[6:2], 2'b10};
    4'b1000 : lower_addr = {rd_addr[6:2], 2'b11};
    default:  lower_addr = 7'b0000000;
  endcase
end

always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      lower_addr_reg <= 0;
    else if(sm_start_resp)
      lower_addr_reg <= lower_addr;
    else if(sm_send_first | sm_send_to_4k | sm_send_max | sm_send_last)
      lower_addr_reg <= 0;
    end

///// Assemble the completion headers
assign completer_id = {BusDev_i, 3'b000};

// write header format
// assign requester_id = {BusDev_i, 3'b000};

assign cpl_header = {8'h4A, 8'h00, 2'h0, attr, 2'h0, pkt_size_reg[11:2],
                     completer_id, 3'b000, 1'b0, remain_bytes[11:0],
                     requester_id, cpl_tag, 1'b0, lower_addr_reg,
32'h00000000};

// ============================================================================
```

```
// Gather the info to generate the PCIe Write header for DMA Wr
// ============================================================================

// In 64-bit mode, calculate the correct space
always @(DmaDstAdr_i) begin
  if (|DmaDstAdr_i[63:32] == 1'b1)
    PCIeAddrSpace_i = 1'b1;  // 64-bit Memory Address
  else
    PCIeAddrSpace_i = 1'b0;  // 32-bit Memory Address
end

// calculate the write dw_len
assign wr_dw_len[9:0] = pkt_size_reg[11:2];

// 64-bit addressing mem write
assign mem_wr64_header = {8'h60, 8'h00, 6'h0, (wr_dw_len[9:0]),
                          completer_id, cpl_tag, 4'hF, fbe,
                       wr_addr_reg[63:32],
                       // wr_addr_reg[31:3], addr_bit2, 2'b00};
                       wr_addr_reg[31:2], 2'b00};

// 32-bit addressing write
assign mem_wr32_header = {8'h40, 8'h00, 6'h0, (wr_dw_len[9:0]),
                          completer_id, cpl_tag, 4'hF, fbe,
                       // wr_addr_reg[31:3], addr_bit2, 2'b00, 32'h00000000};
                       wr_addr_reg[31:2], 2'b00, 32'h00000000};

// muxing the header based on the address decoding space
assign wr_header = PCIeAddrSpace_i ? mem_wr64_header : mem_wr32_header;

// ============================================================================
// command fifo interface
// ============================================================================

assign CmdFifoWrReq_o = sm_send_first | sm_send_last | sm_send_max |
sm_send_to_4k;

// indicate busy one clock before accessing it
assign CmdFifoBusy_o = sm_send_to_4k | sm_send_first | sm_send_max |
sm_send_last;

assign CmdFifoDatin_o = dma_req ? wr_header : cpl_header;

endmodule
```

rx_pcie.v

```
//-----------------------------------------------------------------------------
-
// Title         : rx_pcie
// Project       : PCIe-to-DDR2 SDRAM Reference Design
//-----------------------------------------------------------------------------
-
// File          : rx_pcie.v
// Author        : Altera Corporation
//-----------------------------------------------------------------------------
-
// Functional Description:
// This module is part of the RX application layer interfacing with the PCIe
// controller
//-----------------------------------------------------------------------------
---
//
// Copyright 2003 Altera Corporation. All rights reserved.  Altera products
are
// protected under numerous U.S. and foreign patents, maskwork rights,
copyrights and
// other intellectual property laws.
// This reference design file, and your use thereof, is subject to and
governed by
// the terms and conditions of the applicable Altera Reference Design License
Agreement.
// By using this reference design file, you indicate your acceptance of such
terms and
// conditions between you and Altera Corporation.  In the event that you do
not agree with
// such terms and conditions, you may not use the reference design file.
Please promptly
// destroy any copies you have made.
//
// This reference design file being provided on an "as-is" basis and as an
accommodation
// and therefore all warranties, representations or guarantees of any kind
// (whether express, implied or statutory) including, without limitation,
warranties of
// merchantability, non-infringement, or fitness for a particular purpose, are
// specifically disclaimed.  By making this reference design file available,
Altera
// expressly does not recommend, suggest or require that this reference design
file be
// used in combination with any other product not provided by Altera
// Code was modified by Maxim Leonov
// Copyright 2008 Auckland University of Technology

// turn off bogus verilog processor warnings
// altera message_off 10034 10035 10036 10037 10230

// synthesis translate_off
`timescale 1ns / 1ps
// synthesis translate_on

module rx_pcie

  ( input               Clk_i,
    input               Rstn_i,

    // Rx port interface to PCI Exp core
    input  [135:0]      RxDesc_i,
    input               RxReq_i,
    input               RxDv_i,
    input               RxDfr_i,
    input  [63:0]       RxData_i,
    input  [7:0]        RxBe_i,

    output              RxAck_o,
```

```
  output              RxRetry_o,
  output              RxMask_o,
  output              RxWs_o,
  output              RxAbort_o,

  // Fifo interface
  // *FifoUsedW* signal has 1 extra bit to tell full/empty condition
  input  [5:0]        CmdFifoUsedW_i,  // 32 entries
  output reg          CmdFifoWrReq_o,
  output reg [71:0]   CmdFifoDat_o,

  input  [7:0]        DatFifoUsedW_i,  // 128 entries
  output reg          DatFifoWrReq_o,
  output reg [71:0]   DatFifoDat_o,

  input  [5:0]        PndngRdFifoUsedW_i,
  output              PndgRdFifoWrReq_o,
  output [49:0]       PndgRdHeader_o,

  // DMA interface
  input               DmaBusy_i,
  output              DmaRead_o,
  output              DmaWrite_o,
  output [63:0]       DmaSrcAdr_o,
  output [63:0]       DmaDstAdr_o,
  output [12:0]       DmaByteCnt_o,
  output              DmaStart_o,
  output              Dma_wr_busy_o,
     //*******
     output                    DmaReg_app_enable_o,
     input                     DmaReg_app_finished_i,
     output                    DmaReg_app_finished_o,

  input  [7:0]        TxDaFifo_rdusedw_i,
  // Tx Completion interface
  input               TxCpl_i,
  // this is modified len (+1, +2, or unchanged) (qw)
  input  [9:0]        TxCplLen_i,

    // cfg signals
input      [31:0]                   DevCsr_i,
input      [12:0]                   BusDev_i
);

//state machine encoding
localparam RX_IDLE         = 10'h000;
localparam RX_WR_ACK       = 10'h003;
localparam RX_WR_DATA      = 10'h005;
localparam RX_WR_WAIT      = 10'h009;
localparam RX_RD_ACK       = 10'h011;
localparam RX_RD_RETRY     = 10'h021;
localparam RX_DMA_RD1      = 10'h041;
localparam RX_DMA_RD2      = 10'h081;
localparam RX_DMA_OUT      = 10'h101;
localparam RX_ERR          = 10'h201;


wire                       is_wr_cpl;
wire                       is_rd;
wire  [10:0]               rx_dwlen;
wire  [31:0]               rx_addr;
wire  [7:0]                dma_tag;
wire  [7:0]                cpl_tag;
wire  [11:0]               cpl_bytecount;
wire  [7:0]                rdreq_tag;
wire  [15:0]               requestor_id;
wire  [3:0]                rx_fbe;
wire  [3:0]                rx_lbe;
wire                       len_plus_2;
wire                       len_plus_1;
wire                       dma_len_plus_2;
```

```
wire                    dma_len_plus_1;
wire                    dma_plus_1;
wire                    cmd_fifo_ok;
wire                    dat_fifo_ok;
wire                    pndgrd_fifo_ok;
wire                    rd_cpl_buff_ok;
wire                    dma_cpl_buff_ok;
wire                    rx_wrack;
wire                    rx_wrdata;
wire                    rx_wrwait;
wire                    rx_rdack;
reg                     rx_rdack_reg;
reg                     TxCpl_i_reg;
wire                    rx_rdretry;
wire                    rx_dmard1;
wire                    rx_dmard2;
wire                    rx_dmaout;
reg                     rx_dmard1_reg;
reg                     rx_dmard2_reg;
reg                     rx_dmaout_reg;
wire                    rxidle;
wire                    rx_err;
wire  [71:0]            wr_header;
wire  [71:0]            cpl_header;
wire  [71:0]            rd_header;
wire  [71:0]            dma_header;
wire                    wr_header_sel;
wire                    cpl_header_sel;
wire                    rd_header_sel;
wire                    dma_header_sel;
wire  [3:0]             header_sel;
wire                    is_wr;
wire                    is_cpl_wd;
wire                    is_cpl_wod;
wire                    last_cpl;
wire  [2:0]             cpl_stat;
wire  [6:0]             bar_hit;
wire  [10:0]            txcpl_dw;
wire                    lowbe_mask_sel;
wire                    lowbe_fbe_sel;
wire                    lowbe_lbe_sel;
wire                    highbe_mask_sel;
wire                    highbe_fbe_sel;
wire                    highbe_lbe_sel;
wire                    cpl_success;
wire                    dma_reg_rd;
wire  [1:0]             attr;

reg   [71:0]            rx_header;
reg   [10:0]            rx_modlen; // actual length requested on avalon
wire  [9:0]             dma_qwlen; // actual length requested on avalon
reg   [10:0]            rx_wrdat_cntr;
reg   [11:0]            txcpl_buffer_size;
reg   [11:0]            txcpl_buffer_size_stg;
reg   [11:0]            txdabuf_inc_cnt;
reg   [9:0]             rx_state;
reg   [9:0]             rx_nxt_state;
reg   [3:0]             rx_lowbe;
reg   [3:0]             rx_highbe;
reg                     rxaddr_bit2_reg;
reg                     rxdwlen0_reg;
reg   [10:0]            rx_modlen_reg;
reg   [9:0]             dma_qwlen_reg;
reg   [3:0]             rx_fbe_reg;
reg   [3:0]             rx_lbe_reg;

wire  [10:0]            dma_dwlen;
reg   [10:0]            dma_dwlen_reg;
wire                    bar2_hit;
wire                    dma_reg_wr;
reg   [4:0]             dma_reg_sel;
```

```
  wire                        DmaPloAdr_wen;
  wire                        DmaPhiAdr_wen;
  wire                        DmaDdrAdr_wen;
  wire                        DmaBcnt_wen;
  wire                        DmaCtrl_wen;
  reg                         DmaPloAdr_wen_reg;
  reg                         DmaPhiAdr_wen_reg;
  reg                         DmaDdrAdr_wen_reg;
  reg                         DmaBcnt_wen_reg;
  reg                         DmaCtrl_wen_reg;
  reg   [31:0]                DmaPloAdr_reg;
  reg   [31:0]                DmaPhiAdr_reg;
  reg   [31:0]                DmaDdrAdr_reg;
  reg   [31:0]                DmaBcnt_reg;
  reg   [31:0]                DmaCtrl_reg;
  reg                         DmaStart_reg;

wire          last_rd_segment_1;
wire          last_rd_segment_2;
wire [12:0]   bytes_to_4KB;
wire          to_4KB_sel;
wire          remain_bytes_sel;
wire [9:0]    rd_dw_len;
reg [63:0]                        dma_adr;
reg [63:0]                        dma_adr_reg;
reg [63:0]                        ddr_dma_adr_reg;
wire [12:0]                        rd_size;
reg [12:0]                        rd_size_reg;
wire [12:0]                        remain_bcnt;
reg [12:0]                        remain_bcnt_reg;
reg [12:0]                        remain_rdbytecnt;
reg [12:0]                        remain_rdbytecnt_reg;
reg [12:0]                        byte_size_reg_1;
reg [12:0]                        byte_size_reg_2;
reg [12:0]                        max_rd_size;
reg [12:0]                        max_rd_size_reg;
reg [11:0]                        max_payload_size;
reg [1:0]                         rdsize_sel_reg;
reg [12:0]                        bytes_to_4KB_reg;
reg [7:0]    dma_tag_cntr;
wire          DmaStart_int;
wire          rd_cnt;

  // decoding the rx_desc bus

  assign is_rd
= ~RxDesc_i[126] & (RxDesc_i[124:122]== 3'b000) & ~RxDesc_i[120];
  assign is_wr
= RxDesc_i[126] & (RxDesc_i[124:120]==5'b00000);
  assign is_cpl_wd
= RxDesc_i[126] & (RxDesc_i[124:120]==5'b01010);
  assign is_cpl_wod
= ~RxDesc_i[126] & (RxDesc_i[124:120]==5'b01010);
  assign rx_dwlen
= (RxDesc_i[105:96]==0)? 11'h400 : RxDesc_i[105:96];
  assign rx_addr
= RxDesc_i[125]? RxDesc_i[31:0] : RxDesc_i[63:32];
  assign cpl_tag       = RxDesc_i[47:40];
  assign cpl_bytecount = RxDesc_i[75:64];
  assign rdreq_tag     = RxDesc_i[79:72];
  assign requestor_id  = RxDesc_i[95:80];
  assign rx_fbe        = is_cpl_wd ? 4'hf : RxDesc_i[67:64];
  assign rx_lbe        = is_cpl_wd ? 4'hf : RxDesc_i[71:68];
  assign last_cpl      = (cpl_bytecount[11:2] == rx_dwlen );
  assign cpl_stat      = RxDesc_i[79:77];
  assign cpl_success   = (cpl_stat == 3'b000);
  assign bar_hit       = RxDesc_i[134:128];
  assign bar2_hit      = RxDesc_i[130];
  assign dma_tag       = 8'b11111111;
  assign attr          = RxDesc_i[109:108];
```

```
//****Modification: send enable signal to application****
  assign DmaReg_app_enable_o = DmaCtrl_reg[10];
 assign DmaReg_app_finished_o = DmaCtrl_reg[11];

// decode the max read size

always @(DevCsr_i)
  begin
    case(DevCsr_i[7:5])
      3'b000 : max_rd_size = 128;
      3'b001 : max_rd_size = 256;
      3'b010 : max_rd_size = 512;
      3'b011 : max_rd_size = 1024;
      3'b100 : max_rd_size = 2048;
      default : max_rd_size = 2048;
    endcase
  end

// Need to flop this to fix a timing violation
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i) begin
       max_rd_size_reg <= 13'h0000;
       bytes_to_4KB_reg <= 13'h0000;
    end
    else begin
       max_rd_size_reg <= max_rd_size;
       bytes_to_4KB_reg <= bytes_to_4KB;
    end
  end

// bytes to 4KB boundary
// assign bytes_to_4KB = 13'h1000 - dma_adr_reg[11:0];
assign bytes_to_4KB = 13'h1000 - DmaSrcAdr_o[11:0];

// Divide up the whole dma request to 2 parts if it hits the 4KB boundary
// First set the 1st_byte_size_reg to either the 4KB boundary or
// DmaByteCnt_i
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      byte_size_reg_1 <= 0;
    else if(rxidle & (DmaByteCnt_o > bytes_to_4KB_reg))
      byte_size_reg_1 <= bytes_to_4KB_reg;
    else if(rxidle & (DmaByteCnt_o <= bytes_to_4KB_reg))
      byte_size_reg_1 <= DmaByteCnt_o;
    else if(rx_dmaout & rx_dmard1_reg)
      byte_size_reg_1 <= byte_size_reg_1 - rd_size_reg;
  end

// Divide up the whole dma request to 2 parts if it hits the 4KB boundary
// First set the 2nd_byte_size_reg to either DmaByteCnt_i - bytes_to_4KB
// or 0
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      byte_size_reg_2 <= 0;
    else if(rxidle & (DmaByteCnt_o > bytes_to_4KB_reg))
      byte_size_reg_2 <= (DmaByteCnt_o - bytes_to_4KB_reg);
    else if(rxidle & (DmaByteCnt_o <= bytes_to_4KB_reg))
      byte_size_reg_2 <= 13'h0000;
    else if(rx_dmaout & rx_dmard2_reg)
      byte_size_reg_2 <= byte_size_reg_2 - rd_size_reg;
  end

// read address reg increments byte the amount of byte in each read header
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      ddr_dma_adr_reg <= 0;
    else if(DmaStart_int & rxidle)
```

```
      ddr_dma_adr_reg <= DmaSrcAdr_o;
    else if(rx_dmaout_reg)
      ddr_dma_adr_reg[15:0] <= ddr_dma_adr_reg[15:0] + dma_qwlen_reg;
  end

// read address reg increments byte the amount of byte in each read header
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      dma_adr_reg <= 0;
    else if(DmaStart_int & rxidle)
      dma_adr_reg <= {DmaSrcAdr_o[60:0],3'b000};
    else if(rx_dmaout_reg)
      dma_adr_reg[15:0] <= dma_adr_reg[15:0] + {dma_qwlen_reg,3'b000};
  end

assign remain_bcnt = (byte_size_reg_1 > 13'h0000) ? byte_size_reg_1 :
                     byte_size_reg_2;

assign rd_size = (remain_bcnt >= max_rd_size_reg) ? max_rd_size_reg :
                 remain_bcnt;

// flop this for timing reason, for fmax of 250MHz
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      remain_bcnt_reg <= 0;
    else if (rx_dmard1 | rx_dmard2)
      remain_bcnt_reg <= remain_bcnt;
  end

// flop this for timing reason, for fmax of 250MHz
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      rd_size_reg <= 0;
    else if (rx_dmard1 | rx_dmard2)
      rd_size_reg <= rd_size;
  end

assign last_rd_segment_1 = rx_dmard1_reg & (rd_size_reg == remain_bcnt_reg);
assign last_rd_segment_2 = rx_dmard2_reg & (rd_size_reg == remain_bcnt_reg);

  // modified length from dw to qword length to match the internal 64-bit
datapath
  // to compensate for the misaligned qwords
  //
  // In order to meet timing of 250MHz, some assumptions were made
  // 1. DMA starting address should be qword aligned
  // 2. DMA size should be multiples of 8
  // Because of that, the above logic can be eliminated to improve timing
  // The below is the optimized RTL
  assign dma_dwlen = rd_size_reg[12:2];
  assign dma_qwlen = rd_size_reg[12:3];

  // modified length from dw to qword length to match the internal 64-bit
datapath
  // to compensate for the misaligned qwords
  assign len_plus_2 = rx_addr[2] & ~rx_dwlen[0];
  assign len_plus_1 = rx_dwlen[0];

  always @(len_plus_2, len_plus_1, rx_dwlen)
    begin
      case({len_plus_2,len_plus_1})
        2'b01   : rx_modlen = rx_dwlen + 11'h001;
        2'b10   : rx_modlen = rx_dwlen + 11'h002;
        default : rx_modlen = rx_dwlen;
      endcase
    end

  // counter to track the available buffer size (tx completion) before
```

```
// accepting a read request packet
// TxCplLen_i is the modified length to compensate for the misaligned qwords

// special pipeline signal to solve a timing issue
always @(posedge Clk_i or negedge Rstn_i)  // state machine registers
begin
  if(~Rstn_i) begin
    TxCpl_i_reg <= 1'b0;
    rx_rdack_reg <= 1'b0;
    rx_dmard1_reg <= 1'b0;
    rx_dmard2_reg <= 1'b0;
    rx_dmaout_reg <= 1'b0;
    dma_qwlen_reg <= 0;
    dma_dwlen_reg <= 0;
  end
  else begin
    TxCpl_i_reg <= TxCpl_i;
    rx_rdack_reg <= rx_rdack;
    rx_dmard1_reg <= rx_dmard1;
    rx_dmard2_reg <= rx_dmard2;
    rx_dmaout_reg <= rx_dmaout;
    dma_qwlen_reg <= dma_qwlen;
    dma_dwlen_reg <= dma_dwlen;
  end
end

// assign txcpl_dw = {TxCplLen_i, 1'b0};
assign rd_cnt = ~rx_rdack & ~rx_dmaout;

always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      txdabuf_inc_cnt <= 12'h000;
    else if(TxCpl_i & ~rd_cnt)
      txdabuf_inc_cnt <= txdabuf_inc_cnt + TxCplLen_i;
    else if(~TxCpl_i & rd_cnt)
      txdabuf_inc_cnt <= 12'h000;
    else if(TxCpl_i & rd_cnt)
      txdabuf_inc_cnt <= TxCplLen_i;
    else
      txdabuf_inc_cnt <= txdabuf_inc_cnt;
  end

always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      txcpl_buffer_size <= 12'h060; // 128 QW available
    else if(~rx_rdack & ~rx_dmaout)
      txcpl_buffer_size <= txcpl_buffer_size + txdabuf_inc_cnt;
    else if(rx_rdack & ~rx_dmaout)
      txcpl_buffer_size <= txcpl_buffer_size - rx_modlen_reg[10:1];
    else if(~rx_rdack & rx_dmaout)
      txcpl_buffer_size <= txcpl_buffer_size - dma_qwlen;
    else
      txcpl_buffer_size <= txcpl_buffer_size;
  end

// check buffer space
assign cmd_fifo_ok     = (CmdFifoUsedW_i <= 31);
assign dat_fifo_ok     = (DatFifoUsedW_i <= 125);
assign pndgrd_fifo_ok  = (PndngRdFifoUsedW_i <= 31);
// The correct variable to use should be rx_modlen.
// But to fix a timing violation at 250MHz, rx_dwlen is used
// In any case, the difference between rx_dwlen and rx_modlen can only be
// 1 entry worth of data space in the cpl_buff and so it should not cause
// any functional bug because sufficient buffer is provided in cpl_buff.
// assign rd_cpl_buff_ok  = (txcpl_buffer_size >= {2'b00,rx_modlen[10:1]});
assign rd_cpl_buff_ok  = (txcpl_buffer_size >= {2'b00,rx_dwlen[10:1]});
assign dma_cpl_buff_ok = (txcpl_buffer_size >= {2'b00,dma_qwlen});

// Rx control state machine
```

```
  always @(posedge Clk_i or negedge Rstn_i)  // state machine registers
  begin
    if(~Rstn_i)
      rx_state <= RX_IDLE;
    else
      rx_state <= rx_nxt_state;
  end

  // state machine next state gen
  // treating completion the same as write

always @(*)
  begin
    case(rx_state)
      RX_IDLE :
        if(RxReq_i & (is_wr | is_cpl_wd | is_cpl_wod) & cmd_fifo_ok)
          rx_nxt_state <= RX_WR_ACK;
        else if(RxReq_i & is_rd & cmd_fifo_ok & rd_cpl_buff_ok &
pndgrd_fifo_ok)
          rx_nxt_state <= RX_RD_ACK;
        else if(RxReq_i & is_rd & cmd_fifo_ok & ~pndgrd_fifo_ok)
          rx_nxt_state <= RX_RD_RETRY;
        else if (DmaStart_int & DmaWrite_o & ~DmaBusy_i)
          rx_nxt_state <= RX_DMA_RD1;
        else if (~RxReq_i)
          rx_nxt_state <= RX_IDLE;
        else
          rx_nxt_state <= RX_ERR;

      RX_WR_ACK :
        if(is_cpl_wd | is_wr)
          rx_nxt_state <= RX_WR_DATA;
        else
          rx_nxt_state <= RX_IDLE;
      RX_WR_DATA :
        if(~RxDfr_i & RxDv_i)
          rx_nxt_state <= RX_IDLE;
        else if(~dat_fifo_ok)
          rx_nxt_state <= RX_WR_WAIT;
        else
          rx_nxt_state <= RX_WR_DATA;

      RX_WR_WAIT :
        if(dat_fifo_ok)
          rx_nxt_state <= RX_WR_DATA;
        else
          rx_nxt_state <= RX_WR_WAIT;

      RX_RD_ACK :
          rx_nxt_state <= RX_IDLE;

      RX_RD_RETRY :
          rx_nxt_state <= RX_IDLE;

      RX_DMA_RD1:
       if (cmd_fifo_ok & dma_cpl_buff_ok & pndgrd_fifo_ok)
          rx_nxt_state <= RX_DMA_OUT;
        else
          rx_nxt_state <= RX_DMA_RD1;

      RX_DMA_RD2:
       if (cmd_fifo_ok & dma_cpl_buff_ok & pndgrd_fifo_ok)
          rx_nxt_state <= RX_DMA_OUT;
        else
          rx_nxt_state <= RX_DMA_RD2;

      RX_DMA_OUT:
        if (~last_rd_segment_1 & rx_dmard1_reg)
          rx_nxt_state <= RX_DMA_RD1;
        else if((last_rd_segment_1 & (byte_size_reg_2 == 13'h0000)) ||
```

```
            (last_rd_segment_2))
           rx_nxt_state <= RX_IDLE;
         else if ((last_rd_segment_1 & (byte_size_reg_2 > 13'h0000)) ||
                  (~last_rd_segment_2 & rx_dmard2_reg))
           rx_nxt_state <= RX_DMA_RD2;
        else
           rx_nxt_state <= RX_DMA_OUT;

       RX_ERR:
        if (~RxDfr_i)
           rx_nxt_state <= RX_IDLE;
         else
           rx_nxt_state <= RX_ERR;

       default:
           rx_nxt_state <= RX_IDLE;

       endcase
     end

// state machine output assignments

assign rxidle     = ~rx_state[0];
assign rx_wrack   = rx_state[1];
assign rx_wrdata  = rx_state[2];
assign rx_wrwait  = rx_state[3];
assign rx_rdack   = rx_state[4];
assign rx_rdretry = rx_state[5];
assign rx_dmard1  = rx_state[6];
assign rx_dmard2  = rx_state[7];
assign rx_dmaout  = rx_state[8];
assign rx_err     = rx_state[9];


/// PCI Express core control interface
// assign RxAck_o   = rx_wrack | rx_rdack | rx_err;
assign RxAck_o   = rx_wrack | rx_rdack | (rx_err & RxReq_i);
assign RxRetry_o = rx_rdretry;
assign RxMask_o  = rx_rdretry;
assign RxWs_o    = rx_wrwait | rxidle | rx_wrack ;

// Command and data fifo interface
// the command/data fifo is used to store the selected header information
// and the write/completion data from rx port
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      CmdFifoWrReq_o <= 1'b0;
    else
      CmdFifoWrReq_o <= (rx_wrack & ~dma_reg_wr) | rx_rdack | rx_dmaout;
    end

always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DatFifoWrReq_o <= 1'b0;
    else
      DatFifoWrReq_o <= (RxDv_i & rx_wrdata & ~dma_reg_wr);
    end

// tag generation counter
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      dma_tag_cntr <= 8'h00;
    else if(rxidle)
      dma_tag_cntr <= 8'h00;
    else if(rx_dmaout)
      dma_tag_cntr <= dma_tag_cntr + 1;
  end
```

```
// assemble the various headers
// [4:0]   : Completion tag
// [14:5]  : length in real QW
// [15]    : last completion
// [16]    : completion successful
// [23:17] : BAR hit
// [31:24] : reserved
// [63:32] : Target address
// [64]    : write request packet
// [65]    : read request packet
// [66]    : completion with data
// [67]    : completion without data
// [71:68] : reserved


assign wr_header  = {4'h0, 4'h1, rx_addr, 8'h00, bar_hit[6:0] ,1'b0, 1'b0,
rx_modlen[10:1], 5'h00};
assign rd_header  = {4'h0, 4'h2, rx_addr, 8'h00,bar_hit[6:0], 1'b0, 1'b0,
rx_modlen[10:1], 5'h00};
assign cpl_header = {4'h0, is_cpl_wod, is_cpl_wd, 2'b00, 32'h00000000, 8'h00,
bar_hit[6:0], cpl_success, last_cpl, rx_modlen[10:1], cpl_tag[4:0]};
assign dma_header  = {4'h0, 4'h2, ddr_dma_adr_reg[31:0], 8'h00,bar_hit[6:0],
1'b0, 1'b0, dma_qwlen, dma_tag_cntr[4:0]};

//muxing the header before writting it to the CD buffer
assign wr_header_sel  = is_wr & rx_wrack;
assign cpl_header_sel = (is_cpl_wd | is_cpl_wod) & rx_wrack;
assign rd_header_sel  = rx_rdack;
assign dma_header_sel  = rx_dmaout;

assign header_sel = {dma_header_sel, rd_header_sel, cpl_header_sel,
wr_header_sel};

always @(header_sel, rd_header_sel, cpl_header_sel, wr_header, cpl_header,
rd_header,
         dma_header, dma_header_sel)
  begin
    case(header_sel)
      4'b0001 : rx_header = wr_header;
      4'b0010 : rx_header = cpl_header;
      4'b0100 : rx_header = rd_header;
      4'b1000 : rx_header = dma_header;
      default: rx_header = 72'h0000000000000000;
    endcase
  end


// figuring out the ben

// the rx write data counter
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      rx_wrdat_cntr <= 0;
    else if(rx_wrack)
      rx_wrdat_cntr <= rx_modlen;
    else if((RxDv_i & rx_wrdata))
      rx_wrdat_cntr <= rx_wrdat_cntr - 11'h002;
  end

// registers to hold the needed RxDesc_i fields (that be gone after ack)
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      begin
        rx_fbe_reg <= 0;
        rx_lbe_reg <= 0;
        rxaddr_bit2_reg <= 0;
        rxdwlen0_reg <= 0;
      end
```

```
    else if(rx_wrack)
      begin
        rx_fbe_reg <= rx_fbe;
        rx_lbe_reg <= rx_lbe;
        rxaddr_bit2_reg <= rx_addr[2];
        rxdwlen0_reg <= rx_dwlen[0];
      end
  end

// registers to hold the needed RxDesc_i fields (that be gone after ack)
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      rx_modlen_reg <= 0;
    else
      rx_modlen_reg <=  rx_modlen;
  end

/// the low be [3:0]
// the source can be : 1. masking, 2. first_be, 3. last_be

assign lowbe_mask_sel =  rxaddr_bit2_reg & (rx_wrdat_cntr == rx_modlen_reg);
// the first 64-bit data of the odd address
assign lowbe_fbe_sel  = ~rxaddr_bit2_reg &  (rx_wrdat_cntr == rx_modlen_reg);
// first 64-bit data and even address
assign lowbe_lbe_sel  =  (rxdwlen0_reg ^ rxaddr_bit2_reg) & (rx_wrdat_cntr ==
2);     // at the last data

// muxing the sources
always @(lowbe_lbe_sel, lowbe_fbe_sel, lowbe_mask_sel, rx_fbe_reg, rx_lbe_reg)
  begin
    case({lowbe_lbe_sel, lowbe_fbe_sel, lowbe_mask_sel})
      3'b001 : rx_lowbe = 4'h0;
      3'b010 : rx_lowbe = rx_fbe_reg;
      3'b100 : rx_lowbe = rx_lbe_reg;
      default :  rx_lowbe = 4'hF;
    endcase
  end

/// the high be [7:4]

assign highbe_mask_sel =  (rxdwlen0_reg ^ rxaddr_bit2_reg) & (rx_wrdat_cntr ==
2);
assign highbe_fbe_sel  =   rxaddr_bit2_reg & (rx_wrdat_cntr == rx_modlen_reg);
assign highbe_lbe_sel   =  ~rxdwlen0_reg & ~rxaddr_bit2_reg & (rx_wrdat_cntr
== 2);

// muxing the sources
always @(highbe_lbe_sel, highbe_fbe_sel, highbe_mask_sel, rx_fbe, rx_lbe)
  begin
    case({highbe_lbe_sel, highbe_fbe_sel, highbe_mask_sel})
      3'b001 : rx_highbe = 4'h0;
      3'b010 : rx_highbe = rx_fbe;
      3'b100 : rx_highbe = rx_lbe;
      default :  rx_highbe = 4'hF;
    endcase
  end


/// muxing between the data/ben and the header before writting into the CD
buffer
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      CmdFifoDat_o <= 0;
    else
      CmdFifoDat_o <= rx_header;
  end

always @(posedge Clk_i or negedge Rstn_i)
  begin
```

```
    if(~Rstn_i)
      DatFifoDat_o <= 0;
    else
      DatFifoDat_o <= ({rx_highbe, rx_lowbe, RxData_i});
  end

//// Rx pending read request
// when a read request is accepted and sent to the avalon, some of the header
// info needs to be saved and later used to reconstruct the completion packet

// [7:0]   tag
// [14:8]  7-bit lower address
// [15]    DMA register read
// [31:16] Requestor ID
// [42:32] Requested length
// [46:43] First Ben
// [47]    DMA request
// [49:48] attr


assign PndgRdHeader_o    = rx_rdack ?
       {attr, 1'b0, rx_fbe, rx_dwlen, requestor_id, dma_reg_rd, rx_addr[6:0],
rdreq_tag} :
       {attr, 1'b1, 4'b1111, dma_dwlen, requestor_id, 1'b0, dma_adr_reg[6:0],
dma_tag_cntr};
assign PndgRdFifoWrReq_o = rx_rdack | rx_dmaout;

// if bar2_hit and a wr_req
// decode the lower bits of the address
// find out which dma register it is accessing
// write the data to the dma register
// when the dma start register is written
// output the dma request

assign dma_reg_wr = is_wr & bar2_hit;
assign dma_reg_rd = is_rd & bar2_hit;

always @(rx_addr)
  begin
    case(rx_addr[7:0])
      8'b00000000 : dma_reg_sel = 5'b00001;     // addr[31:0] of PCIe Addr
      8'b00000100 : dma_reg_sel = 5'b00010;     // addr[63:32] of PCIe Addr
      8'b00001000 : dma_reg_sel = 5'b00100;     // DmaByteCnt
      8'b00001100 : dma_reg_sel = 5'b01000;     // DmaCtrl
      8'b00010100 : dma_reg_sel = 5'b10000;     // addr[31:0] of DDR2 Addr
      default     : dma_reg_sel = 5'b00000;
    endcase
  end

assign DmaPloAdr_wen = dma_reg_wr & dma_reg_sel[0];
assign DmaPhiAdr_wen = dma_reg_wr & dma_reg_sel[1];
assign DmaBcnt_wen = dma_reg_wr & dma_reg_sel[2];
assign DmaCtrl_wen = dma_reg_wr & dma_reg_sel[3];
assign DmaDdrAdr_wen = dma_reg_wr & dma_reg_sel[4];

always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      begin
        DmaPloAdr_wen_reg <= 0;
        DmaPhiAdr_wen_reg <= 0;
        DmaDdrAdr_wen_reg <= 0;
        DmaBcnt_wen_reg <= 0;
        DmaCtrl_wen_reg <= 0;
      end
    else if(rx_wrack)
      begin
        DmaPloAdr_wen_reg <= DmaPloAdr_wen;
        DmaPhiAdr_wen_reg <= DmaPhiAdr_wen;
        DmaDdrAdr_wen_reg <= DmaDdrAdr_wen;
        DmaBcnt_wen_reg <= DmaBcnt_wen;
```

```
        DmaCtrl_wen_reg <= DmaCtrl_wen;
      end
  end

// DmaPloAdr - Dma PCIe address lower half, i.e. [31:0]
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaPloAdr_reg <= 0;
    else if(rx_wrdata & DmaPloAdr_wen_reg)
      DmaPloAdr_reg <= RxData_i[31:0];
  end

// DmaPhiAdr - Dma PCIe address upper half, i.e. [63:32]
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaPhiAdr_reg <= 0;
    else if(rx_wrdata & DmaPhiAdr_wen_reg)
      DmaPhiAdr_reg <= RxData_i[31:0];
  end

// DmaDdrAdr
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaDdrAdr_reg <= 0;
    else if(rx_wrdata & DmaDdrAdr_wen_reg)
      DmaDdrAdr_reg <= RxData_i[31:0];
  end

// Dma Byte Cnt[12:0].  Only [12:0] is used
// The rest is ignored.
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaBcnt_reg <= 0;
    else if(rx_wrdata & DmaBcnt_wen_reg)
      DmaBcnt_reg <= RxData_i[31:0];
  end

// DmaWr when DmaBcnt_reg[6:5] = 10b (3DW) or 11b (4DW)
// DmaRd when DmaBcnt_reg[6:5] = 00b (3DW) or 01b (4DW)
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaCtrl_reg[6:5] <= 0;
    else if(rx_wrdata & DmaCtrl_wen_reg)
      DmaCtrl_reg[6:5] <= RxData_i[6:5];
  end

//**** Modification made to check ***************

always @(posedge Clk_i or negedge Rstn_i)
 begin
   if(~Rstn_i)
     DmaCtrl_reg[10] <= 1'b0;
   else if(rx_wrdata & DmaCtrl_wen_reg)
     DmaCtrl_reg[10] <= RxData_i[10];
end

always @(posedge Clk_i or negedge Rstn_i)
 begin
   if(~Rstn_i)
     DmaCtrl_reg[11] <= 1'b0;
   else if(DmaReg_app_finished_i)
     DmaCtrl_reg[11] <= DmaReg_app_finished_i;
 end
//***********************************************

// DmaStart bit can only set by user at DmaCtrl_reg[0]
```

```
// DmaDone bit can only set by hardware at DmaCtrl_reg[8]
// When DmaStart bit is written, Dma will start if DmaBusy is not asserted
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaCtrl_reg[7] <= 1'b0;
    else if(DmaStart_reg)
      DmaCtrl_reg[7] <= 1'b0;
    else if(rx_wrdata & DmaCtrl_wen_reg)
      DmaCtrl_reg[7] <= RxData_i[7];
  end


always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaStart_reg <= 0;
    else
      DmaStart_reg <= DmaCtrl_reg[7];
  end

assign DmaRead_o = ~DmaCtrl_reg[6];
assign DmaWrite_o = DmaCtrl_reg[6];
assign DmaSrcAdr_o = DmaWrite_o ? {32'h00000000,DmaDdrAdr_reg}
                                : {DmaPhiAdr_reg,DmaPloAdr_reg};
assign DmaDstAdr_o = DmaWrite_o ? {DmaPhiAdr_reg,DmaPloAdr_reg}
                                : {32'h00000000,DmaDdrAdr_reg};
assign DmaByteCnt_o = DmaBcnt_reg[12:0];
assign DmaStart_o = DmaCtrl_reg[7];
assign DmaStart_int = DmaCtrl_reg[7] & ~DmaStart_reg;
assign Dma_wr_busy_o = rx_dmard1 | rx_dmard2 | rx_dmaout;

// assign DmaCtrl_reg[31] = 1 when DmaBusy from top level
// assign DmaCtrl_reg[31] = 0 when DmaDone from top level
always @(posedge Clk_i or negedge Rstn_i)
  begin
    if(~Rstn_i)
      DmaCtrl_reg[31] <= 1'b0;
    else
      DmaCtrl_reg[31] <= DmaBusy_i;
  end


endmodule
```

Appendix A7. The Switch

switch_top.v

```
// Code was created by Maxim Leonov
// Copyright 2008 Auckland University of Technology
module switch_top

  (
    input         Clk_i,
    input         Rstn_i,

       input           switch_select_i,
       output          switch_select_o,

    //rx_top interfaces
    input         rx_top_read_i,
    input         rx_top_write_i,
    input [31:0]  rx_top_address_i,
    input [63:0]  rx_top_write_data_i,
    input [7:0]   rx_top_byte_enable_i,
    input [1:0]   rx_top_burst_count_i,

       output        rx_top_ready_o,
    output        rx_top_wrdata_req_o,

       //tx_top interfaces
       output          tx_top_rd_data_valid_o,
       output [63:0] tx_top_read_data_o,

       //driver interfaces
    input         driver_read_i,
    input         driver_write_i,
    input [1:0]   driver_bank_address_i,
    input [9:0]   driver_col_address_i,
    input [12:0]  driver_row_address_i,
    input [63:0]  driver_write_data_i,
    input [7:0]   driver_byte_enable_i,
    input [1:0]   driver_burst_count_i,

       output        driver_ready_o,
    output        driver_wrdata_req_o,
    output [63:0] driver_read_data_o,
       output          driver_rdata_valid_o,

    //ddr_ctrlr interfaces
    input          DdrReady_i,//1
    input          DdrWrDataReq_i,//2
    input [63:0]   DdrReadData_i,//3
       input           DddReadDataValid_i,

       output         DdrRead_o,//5
       output         DdrWrite_o,//6
    output [31:0]  DdrAddress_o,//7
    output [63:0]  DdrWriteData_o,//8
    output [7:0]   DdrByteEnable_o,//9
    output [1:0]   DdrBurstCount_o  //10

  );


wire               driver_read;
wire               driver_write;
wire [31:0]        driver_address;
wire [63:0]        driver_write_data;
```

```
wire [7:0]          driver_byte_enable;
wire [1:0]          driver_burst_count;
wire                driver_ddr_ready;
wire                driver_wrdata_req;

wire                rx_top_read;
wire                rx_top_write;
wire [31:0]         rx_top_address;
wire [63:0]         rx_top_write_data;
wire [7:0]          rx_top_byte_enable;
wire [1:0]          rx_top_burst_count;
wire                rx_top_ready;
wire                rx_top_wrdata_req;

assign switch_select_o = switch_select_i;

//connect up the address bits
assign driver_address[8 : 0] = driver_col_address_i[9 : 1];
assign driver_address[21: 9] = driver_row_address_i;
assign driver_address[23:22] = driver_bank_address_i;
assign driver_address[31:24] = 8'h00;

assign DdrRead_o = switch_select_i? driver_read_i : rx_top_read_i;
assign DdrWrite_o = switch_select_i ? driver_write_i : rx_top_write_i;
assign DdrAddress_o = switch_select_i ? driver_address : rx_top_address_i;
assign DdrWriteData_o = switch_select_i ?  driver_write_data_i:
rx_top_write_data_i;
assign DdrByteEnable_o = switch_select_i ? driver_byte_enable_i :
rx_top_byte_enable_i;
assign DdrBurstCount_o = switch_select_i ? driver_burst_count_i :
rx_top_burst_count_i;

assign driver_ready_o = switch_select_i ? DdrReady_i : 1'b0;
assign rx_top_ready_o = ~switch_select_i ? DdrReady_i : 1'b0;

assign driver_wrdata_req_o = switch_select_i ? DdrWrDataReq_i : 1'b0;
assign rx_top_wrdata_req_o = ~switch_select_i ? DdrWrDataReq_i : 1'b0;

assign driver_read_data_o = switch_select_i ? DdrReadData_i : {64{1'b0}};
assign tx_top_read_data_o = ~switch_select_i ? DdrReadData_i : {64{1'b0}};

assign driver_rdata_valid_o = switch_select_i ? DddReadDataValid_i : 1'b0;
assign tx_top_rd_data_valid_o = ~switch_select_i ? DddReadDataValid_i : 1'b0;

endmodule
```

Appendix A8. The Correlator's DDR Controller Driver

ddr_ctrl_driver.v

```
// Code was created by Maxim Leonov
// Copyright 2008 Auckland University of Technology

// synthesis translate_off
`timescale 1ns / 1ps
// synthesis translate_on

module ddr_ctrl_driver (
      // inputs:
       clk,
       local_rdata,
       local_rdata_valid,
       local_ready,
       local_wdata_req,
       reset_n,

       enable,
       data_r1,
       data_r2,
       data_r3,
       data_r4,
       data_r5,
       data_r6,
       data_r7,
       data_r8,
       data_r9,
       data_r10,
       data_r11,
       data_r12,
       data_r13,
       data_r14,
       data_r15,

       in_stream_x1_ready,
       in_stream_x2_ready,
       in_stream_x3_ready,
       in_stream_x4_ready,
       in_stream_x5_ready,
       in_stream_x6_ready,
       out_stream_r1_ready,
       out_stream_r2_ready,
       out_stream_r3_ready,
       out_stream_r4_ready,
       out_stream_r5_ready,
       out_stream_r6_ready,
       out_stream_r7_ready,
       out_stream_r8_ready,
       out_stream_r9_ready,
       out_stream_r10_ready,
       out_stream_r11_ready,
       out_stream_r12_ready,
       out_stream_r13_ready,
       out_stream_r14_ready,
       out_stream_r15_ready,
       out_stream_r1_closed,
       out_stream_r2_closed,
       out_stream_r3_closed,
       out_stream_r4_closed,
       out_stream_r5_closed,
       out_stream_r6_closed,
       out_stream_r7_closed,
```

```
      out_stream_r8_closed,
      out_stream_r9_closed,
      out_stream_r10_closed,
      out_stream_r11_closed,
      out_stream_r12_closed,
      out_stream_r13_closed,
      out_stream_r14_closed,
      out_stream_r15_closed,


      // outputs:
      data_to_corr_o,
      corr_x1_rd_data_valid_o,
      corr_x2_rd_data_valid_o,
      corr_x3_rd_data_valid_o,
      corr_x4_rd_data_valid_o,
      corr_x5_rd_data_valid_o,
      corr_x6_rd_data_valid_o,
      corr_wrdata_req_o_r1,
      corr_wrdata_req_o_r2,
      corr_wrdata_req_o_r3,
      corr_wrdata_req_o_r4,
      corr_wrdata_req_o_r5,
      corr_wrdata_req_o_r6,
      corr_wrdata_req_o_r7,
      corr_wrdata_req_o_r8,
      corr_wrdata_req_o_r9,
      corr_wrdata_req_o_r10,
      corr_wrdata_req_o_r11,
      corr_wrdata_req_o_r12,
      corr_wrdata_req_o_r13,
      corr_wrdata_req_o_r14,
      corr_wrdata_req_o_r15,
      read_count_o,

      burst_begin,
      local_bank_addr,
      local_be,
      local_col_addr,
      local_cs_addr,
      local_read_req,
      local_row_addr,
      local_size,
      local_wdata,
      local_write_req,
      pnf_per_byte,
      pnf_persist,
      test_complete
                                  );

output           burst_begin;
output  [  1: 0] local_bank_addr;
output  [  7: 0] local_be;
output  [  9: 0] local_col_addr;
output           local_cs_addr;
output           local_read_req;
output  [ 12: 0] local_row_addr;
output  [  1: 0] local_size;
output  [ 63: 0] local_wdata;
output           local_write_req;
output  [  7: 0] pnf_per_byte;
output           pnf_persist;
output           test_complete;
output  [ 63: 0] data_to_corr_o;
output               corr_x1_rd_data_valid_o;
output               corr_x2_rd_data_valid_o;
output               corr_x3_rd_data_valid_o;
output               corr_x4_rd_data_valid_o;
output               corr_x5_rd_data_valid_o;
output               corr_x6_rd_data_valid_o;
output               corr_wrdata_req_o_r1;
```

```
  output                corr_wrdata_req_o_r2;
  output                corr_wrdata_req_o_r3;
  output                corr_wrdata_req_o_r4;
  output                corr_wrdata_req_o_r5;
  output                corr_wrdata_req_o_r6;
  output                corr_wrdata_req_o_r7;
  output                corr_wrdata_req_o_r8;
  output                corr_wrdata_req_o_r9;
  output                corr_wrdata_req_o_r10;
  output                corr_wrdata_req_o_r11;
  output                corr_wrdata_req_o_r12;
  output                corr_wrdata_req_o_r13;
  output                corr_wrdata_req_o_r14;
  output                corr_wrdata_req_o_r15;
  output  [ 15: 0] read_count_o;


  input             clk;
input                       enable;
input                       in_stream_x1_ready;
input                       in_stream_x2_ready;
input                       in_stream_x3_ready;
input                       in_stream_x4_ready;
input                       in_stream_x5_ready;
input                       in_stream_x6_ready;
input                       out_stream_r1_ready;
input                       out_stream_r2_ready;
input                       out_stream_r3_ready;
input                       out_stream_r4_ready;
input                       out_stream_r5_ready;
input                       out_stream_r6_ready;
input                       out_stream_r7_ready;
input                       out_stream_r8_ready;
input                       out_stream_r9_ready;
input                       out_stream_r10_ready;
input                       out_stream_r11_ready;
input                       out_stream_r12_ready;
input                       out_stream_r13_ready;
input                       out_stream_r14_ready;
input                       out_stream_r15_ready;
input                       out_stream_r1_closed;
input                       out_stream_r2_closed;
input                       out_stream_r3_closed;
input                       out_stream_r4_closed;
input                       out_stream_r5_closed;
input                       out_stream_r6_closed;
input                       out_stream_r7_closed;
input                       out_stream_r8_closed;
input                       out_stream_r9_closed;
input                       out_stream_r10_closed;
input                       out_stream_r11_closed;
input                       out_stream_r12_closed;
input                       out_stream_r13_closed;
input                       out_stream_r14_closed;
input                       out_stream_r15_closed;
input   [ 63: 0]    local_rdata;
input                       local_rdata_valid;
input                       local_ready;
input                       local_wdata_req;
input                       reset_n;
input  [ 63: 0]     data_r1;
input  [ 63: 0]     data_r2;
input  [ 63: 0]     data_r3;
input  [ 63: 0]     data_r4;
input  [ 63: 0]     data_r5;
input  [ 63: 0]     data_r6;
input  [ 63: 0]     data_r7;
input  [ 63: 0]     data_r8;
input  [ 63: 0]     data_r9;
input  [ 63: 0]     data_r10;
input  [ 63: 0]     data_r11;
```

```
input  [ 63: 0]      data_r12;
input  [ 63: 0]      data_r13;
input  [ 63: 0]      data_r14;
input  [ 63: 0]      data_r15;



  wire    [  1: 0] LOCAL_BURST_LEN_s;
  wire    [  1: 0] MAX_BANK;
  wire             MAX_CHIPSEL;
  wire    [  9: 0] MAX_COL;
  wire    [ 12: 0] MAX_ROW;
  wire             MIN_CHIPSEL;
  wire    [  1: 0] DATA_BANK;
  wire    [  9: 0] DATA_COL;
  wire    [ 12: 0] DATA_ROW;
  wire             DATA_CHIPSEL;
  wire    [  1: 0] DATA_OUT_BANK;
  wire    [  9: 0] DATA_OUT_COL;
  wire    [ 12: 0] DATA_OUT_ROW;
  wire             DATA_OUT_CHIPSEL;
  wire             avalon_burst_mode;
  wire             avalon_read_burst_max_address;
  reg     [  1: 0] bank_addr;
//  wire    [ 17: 0] be;
  wire    [  7: 0] be;
  reg     [  2: 0] burst_beat_count;
  reg              burst_begin;
  reg     [  9: 0] col_addr;
  wire    [  7: 0] compare;
  reg     [  7: 0] compare_reg;
  reg     [  7: 0] compare_valid;
  reg     [  7: 0] compare_valid_reg;
  reg              cs_addr;
  wire    [63: 0] dgen_data;
  reg              dgen_enable;
  reg     [63: 0] dgen_ldata;
  reg              dgen_load;
  wire             dgen_pause;
  reg              last_rdata_valid;
  reg              last_wdata_req;
  wire    [  1: 0] local_bank_addr;
//  wire    [ 17: 0] local_be;
  wire    [  7: 0] local_be;
  wire    [  9: 0] local_col_addr;
  wire             local_cs_addr;
  wire             local_read_req;
  wire    [ 12: 0] local_row_addr;
  wire    [  1: 0] local_size;
//  wire    [143: 0] local_wdata;
  wire    [63: 0] local_wdata;
  wire             local_write_req;
  wire    [ 17: 0] pnf_per_byte;
  reg              pnf_persist;
  reg              pnf_persist1;
  wire             reached_data_address;
  reg              reached_data_count;
  wire             reached_data_out_address;
  reg              reached_data_out_count;
  wire             reached_max_address;
  reg              reached_max_count;
  reg              read_req;
  reg     [  7: 0] reads_remaining;
  reg              reset_address;
  reg     [ 12: 0] row_addr;
  wire    [  1: 0] size;
  reg     [  3: 0] state;
  reg              test_complete;
  reg              wait_first_write_data;
//  wire    [143: 0] wdata;
  wire    [63: 0] wdata;
```

```
  wire             wdata_req;
  reg              write_req;
  reg     [  7: 0] writes_remaining;

wire                      enable;
wire                      in_streams_ready;

reg                             out_stream_ready;
reg                             out_stream_closed;

reg          [ 15: 0]     read_count;

reg          [  3: 0]     read_state;
reg          [  3: 0]     write_state;

reg                             read_done;

reg                             stream_x1_wr_enable_reg;
reg                             stream_x2_wr_enable_reg;
reg                             stream_x3_wr_enable_reg;
reg                             stream_x4_wr_enable_reg;
reg                             stream_x5_wr_enable_reg;
reg                             stream_x6_wr_enable_reg;

reg     corr_wrdata_req_r0_reg;
reg     corr_wrdata_req_r1_reg;
reg     corr_wrdata_req_r2_reg;
reg     corr_wrdata_req_r3_reg;
reg     corr_wrdata_req_r4_reg;
reg     corr_wrdata_req_r5_reg;
reg     corr_wrdata_req_r6_reg;
reg     corr_wrdata_req_r7_reg;
reg     corr_wrdata_req_r8_reg;
reg     corr_wrdata_req_r9_reg;
reg     corr_wrdata_req_r10_reg;
reg     corr_wrdata_req_r11_reg;
reg     corr_wrdata_req_r12_reg;
reg     corr_wrdata_req_r13_reg;
reg     corr_wrdata_req_r14_reg;

assign corr_x1_rd_data_valid_o = stream_x1_wr_enable_reg;
assign corr_x2_rd_data_valid_o = stream_x2_wr_enable_reg;
assign corr_x3_rd_data_valid_o = stream_x3_wr_enable_reg;
assign corr_x4_rd_data_valid_o = stream_x4_wr_enable_reg;
assign corr_x5_rd_data_valid_o = stream_x5_wr_enable_reg;
assign corr_x6_rd_data_valid_o = stream_x6_wr_enable_reg;

assign corr_wrdata_req_o_r1 = corr_wrdata_req_r0_reg;
assign corr_wrdata_req_o_r2 = corr_wrdata_req_r1_reg;
assign corr_wrdata_req_o_r3 = corr_wrdata_req_r2_reg;
assign corr_wrdata_req_o_r4 = corr_wrdata_req_r3_reg;
assign corr_wrdata_req_o_r5 = corr_wrdata_req_r4_reg;
assign corr_wrdata_req_o_r6 = corr_wrdata_req_r5_reg;
assign corr_wrdata_req_o_r7 = corr_wrdata_req_r6_reg;
assign corr_wrdata_req_o_r8 = corr_wrdata_req_r7_reg;
assign corr_wrdata_req_o_r9 = corr_wrdata_req_r8_reg;
assign corr_wrdata_req_o_r10 = corr_wrdata_req_r9_reg;
assign corr_wrdata_req_o_r11 = corr_wrdata_req_r10_reg;
assign corr_wrdata_req_o_r12 = corr_wrdata_req_r11_reg;
assign corr_wrdata_req_o_r13 = corr_wrdata_req_r12_reg;
assign corr_wrdata_req_o_r14 = corr_wrdata_req_r13_reg;
assign corr_wrdata_req_o_r15 = corr_wrdata_req_r14_reg;

  assign avalon_burst_mode = 0;
  assign MIN_CHIPSEL = 0;
  assign MAX_CHIPSEL = 0;
  assign MAX_ROW = 8;//0;
  assign MAX_BANK = 0;//0;
//  assign MAX_COL = 16;//96;
  //
//     assign MAX_ROW = 1<<(13-1);
```

```
       assign MAX_COL = 1<<(10-1);

       assign DATA_CHIPSEL = 0;
       assign DATA_BANK = 0;
//     assign DATA_ROW = 1<<(4-1);
       assign DATA_COL = 1<<(10-1);
//     assign DATA_ROW = 72;
       assign DATA_ROW = 200;

       assign DATA_OUT_CHIPSEL = 0;
       assign DATA_OUT_BANK = 0;
//     assign DATA_OUT_ROW = 4;
       assign DATA_OUT_ROW = 0;
       assign DATA_OUT_COL = 132;//32 output samples

//     assign DATA_OUT_COL = 512;

  assign local_cs_addr = cs_addr;

  assign local_row_addr = row_addr;
  assign local_bank_addr = bank_addr;
  assign local_col_addr = col_addr;
  assign local_write_req = write_req;
  assign local_read_req = read_req;
  assign local_wdata = wdata;

assign data_to_corr_o = local_rdata;
//assign corr_rd_data_valid_o = local_rdata_valid;
//assign wdata = data_from_corr_i;

assign read_count_o = read_count;
  //The LOCAL_BURST_LEN_s is a signal used insted of the parameter
LOCAL_BURST_LEN
  assign LOCAL_BURST_LEN_s = 1;
  //LOCAL INTERFACE (NON-AVALON)
  assign wdata_req = local_wdata_req;
//assign corr_wrdata_req_o = local_wdata_req;


  assign local_be = be;

  assign be = -1;
  assign pnf_per_byte = compare_valid_reg;
  assign local_size = size;
  //FIX
  assign size = LOCAL_BURST_LEN_s[1 : 0];
  assign reached_data_address = (col_addr >= (DATA_COL - (2 * 2))) &&
(row_addr == DATA_ROW) && (bank_addr == DATA_BANK) && (cs_addr ==
DATA_CHIPSEL);
  assign reached_data_out_address = (col_addr >= (DATA_OUT_COL - (2 * 2))) &&
(row_addr == DATA_OUT_ROW) && (bank_addr == DATA_OUT_BANK) && (cs_addr ==
DATA_OUT_CHIPSEL);
  assign reached_max_address = (col_addr >= (MAX_COL - (2 * 2))) && (row_addr
== MAX_ROW) && (bank_addr == MAX_BANK) && (cs_addr == MAX_CHIPSEL);
  assign avalon_read_burst_max_address = (col_addr >= (MAX_COL - (2 * 4))) &&
(row_addr == MAX_ROW) && (bank_addr == MAX_BANK) && (cs_addr == MAX_CHIPSEL);

one_bit_mux  out_ready_mux (
       .data0 ( out_stream_r1_ready ),
       .data1 ( out_stream_r2_ready ),
       .data10 ( out_stream_r11_ready ),
       .data11 ( out_stream_r12_ready ),
       .data12 ( out_stream_r13_ready ),
       .data13 ( out_stream_r14_ready ),
       .data14 ( out_stream_r15_ready ),
       .data2 ( out_stream_r3_ready ),
       .data3 ( out_stream_r4_ready ),
       .data4 ( out_stream_r5_ready ),
       .data5 ( out_stream_r6_ready ),
       .data6 ( out_stream_r7_ready ),
       .data7 ( out_stream_r8_ready ),
```

```
       .data8 ( out_stream_r9_ready ),
       .data9 ( out_stream_r10_ready ),
       .sel ( write_state ),
       .result ( out_stream_ready )
       );

one_bit_mux  out_closed_mux (
       .data0 ( out_stream_r1_closed ),
       .data1 ( out_stream_r2_closed ),
       .data10 ( out_stream_r11_closed ),
       .data11 ( out_stream_r12_closed ),
       .data12 ( out_stream_r13_closed ),
       .data13 ( out_stream_r14_closed ),
       .data14 ( out_stream_r15_closed ),
       .data2 ( out_stream_r3_closed ),
       .data3 ( out_stream_r4_closed ),
       .data4 ( out_stream_r5_closed ),
       .data5 ( out_stream_r6_closed ),
       .data6 ( out_stream_r7_closed ),
       .data7 ( out_stream_r8_closed ),
       .data8 ( out_stream_r9_closed ),
       .data9 ( out_stream_r10_closed ),
       .sel ( write_state ),
       .result ( out_stream_closed )
       );

mux_64bits   out_data_mux (
       .data0x ( data_r1 ),
       .data1x ( data_r2 ),
       .data10x ( data_r11 ),
       .data11x ( data_r12 ),
       .data12x ( data_r13 ),
       .data13x ( data_r14 ),
       .data14x ( data_r15 ),
       .data2x ( data_r3 ),
       .data3x ( data_r4 ),
       .data4x ( data_r5 ),
       .data5x ( data_r6 ),
       .data6x ( data_r7 ),
       .data7x ( data_r8 ),
       .data8x ( data_r9 ),
       .data9x ( data_r10 ),
       .sel ( write_state ),
       .result ( wdata )
       );

  //
  //------------------------------------------------------------------
  //Main clocked process
  //------------------------------------------------------------------
  //Read / Write control state machine & address counter
  //------------------------------------------------------------------
  always @(posedge clk or negedge reset_n)
    begin
      if (reset_n == 0)
        begin
          //Reset - asynchronously force all register outputs LOW
          state <= 4'b0000;
                read_state <= 4'b0000;
                write_state <= 4'b0000;
                read_done <= 1'b0;

          write_req <= 1'b0;
          read_req <= 1'b0;
          burst_begin <= 0;
          burst_beat_count <= 0;
          cs_addr <= 0;
          row_addr <= 0;
          bank_addr <= 0;
          col_addr <= 0;
          dgen_enable <= 1'b0;
```

```
          dgen_load <= 1'b0;
          wait_first_write_data <= 1'b0;
               reached_data_count <= 1'b0;
               reached_data_out_count <= 1'b0;
               reached_max_count <= 1'b0;
          test_complete <= 1'b0;
          writes_remaining <= 0;
          reads_remaining <= 0;
          reset_address <= 1'b0;
               read_count <= 0;
               stream_x1_wr_enable_reg <= 1'b0;
               stream_x2_wr_enable_reg <= 1'b0;
               stream_x3_wr_enable_reg <= 1'b0;
               stream_x4_wr_enable_reg <= 1'b0;
               stream_x5_wr_enable_reg <= 1'b0;
               stream_x6_wr_enable_reg <= 1'b0;
             corr_wrdata_req_r0_reg <= 1'b0;
             corr_wrdata_req_r1_reg <= 1'b0;
             corr_wrdata_req_r2_reg <= 1'b0;
             corr_wrdata_req_r3_reg <= 1'b0;
             corr_wrdata_req_r4_reg <= 1'b0;
             corr_wrdata_req_r5_reg <= 1'b0;
             corr_wrdata_req_r6_reg <= 1'b0;
             corr_wrdata_req_r7_reg <= 1'b0;
             corr_wrdata_req_r8_reg <= 1'b0;
             corr_wrdata_req_r9_reg <= 1'b0;
             corr_wrdata_req_r10_reg <= 1'b0;
             corr_wrdata_req_r11_reg <= 1'b0;
             corr_wrdata_req_r12_reg <= 1'b0;
             corr_wrdata_req_r13_reg <= 1'b0;
             corr_wrdata_req_r14_reg <= 1'b0;

//             out_stream_ready <= 1'b0;
//             out_stream_closed <= 1'b0;
        end
      else if(enable)
        begin
          reset_address <= 1'b0;
          reached_max_count <= reached_max_address;
               reached_data_count <= reached_data_address;
               reached_data_out_count <= reached_data_out_address;
          read_req <= 1'b0;
          write_req <= 1'b0;
          dgen_load <= 1'b0;
               stream_x1_wr_enable_reg <= 1'b0;
               stream_x2_wr_enable_reg <= 1'b0;
               stream_x3_wr_enable_reg <= 1'b0;
               stream_x4_wr_enable_reg <= 1'b0;
               stream_x5_wr_enable_reg <= 1'b0;
               stream_x6_wr_enable_reg <= 1'b0;
             corr_wrdata_req_r0_reg <= 1'b0;
             corr_wrdata_req_r1_reg <= 1'b0;
             corr_wrdata_req_r2_reg <= 1'b0;
             corr_wrdata_req_r3_reg <= 1'b0;
             corr_wrdata_req_r4_reg <= 1'b0;
             corr_wrdata_req_r5_reg <= 1'b0;
             corr_wrdata_req_r6_reg <= 1'b0;
             corr_wrdata_req_r7_reg <= 1'b0;
             corr_wrdata_req_r8_reg <= 1'b0;
             corr_wrdata_req_r9_reg <= 1'b0;
             corr_wrdata_req_r10_reg <= 1'b0;
             corr_wrdata_req_r11_reg <= 1'b0;
             corr_wrdata_req_r12_reg <= 1'b0;
             corr_wrdata_req_r13_reg <= 1'b0;
             corr_wrdata_req_r14_reg <= 1'b0;
//             read_done <= 1'b0;
//             read_count <= 0;
//           test_complete <= 1'b0;
          if (last_wdata_req)
              wait_first_write_data <= 0;
          if (write_req && local_ready)
```

```
            begin
              if (wdata_req)
                  writes_remaining <= writes_remaining + (size - 1);
              else
                writes_remaining <= writes_remaining + size;
            end
          else if ((wdata_req) && (writes_remaining > 0))
              //size
              writes_remaining <= writes_remaining - 1'b1;
          else
            writes_remaining <= writes_remaining;
          if (read_req && local_ready)
            begin
              if (local_rdata_valid)
                  reads_remaining <= reads_remaining + (size - 1);
              else
                reads_remaining <= reads_remaining + size;
            end
          else if ((local_rdata_valid) && (reads_remaining > 0))
              reads_remaining <= reads_remaining - 1'b1;
          else
            reads_remaining <= reads_remaining;
          case (state)

              4'd0: begin
                  reached_max_count <= 0;
                            reached_data_count <= 0;
                            reached_data_out_count <= 1'b0;
                  if (avalon_burst_mode == 0)
                    begin
                      if (1 == 0)
                          state <= 5;
                      else
                        state <= 1;
                    end
                  else
                    begin
                      burst_begin <= 1;
                      write_req <= 1'b1;
                      state <= 10;
                    end

                  //Reset just in case!
                  writes_remaining <= 0;

                  reads_remaining <= 0;
              end // 4'd0

              4'd3: begin
                                    case (write_state)
       4'd0: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 1;
       reset_address <= 1'b1;
       corr_wrdata_req_r0_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r0_reg <= local_wdata_req;
       end
       end //write_state 4'd0
       4'd1: begin
if (reached_data_out_count)
```

```
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 2;
       reset_address <= 1'b1;
       corr_wrdata_req_r1_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r1_reg <= local_wdata_req;
       end
       end //write_state 4'd1
       4'd2: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 3;
       reset_address <= 1'b1;
       corr_wrdata_req_r2_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r2_reg <= local_wdata_req;
       end
       end //write_state 4'd2
       4'd3: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 4;
       reset_address <= 1'b1;
       corr_wrdata_req_r3_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r3_reg <= local_wdata_req;
       end
       end //write_state 4'd3
       4'd4: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 5;
       reset_address <= 1'b1;
       corr_wrdata_req_r4_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
```

```
corr_wrdata_req_r4_reg <= local_wdata_req;
       end
       end //write_state 4'd4
       4'd5: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 6;
       reset_address <= 1'b1;
       corr_wrdata_req_r5_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r5_reg <= local_wdata_req;
       end
       end //write_state 4'd5
       4'd6: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 7;
       reset_address <= 1'b1;
       corr_wrdata_req_r6_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r6_reg <= local_wdata_req;
       end
       end //write_state 4'd6
       4'd7: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 8;
       reset_address <= 1'b1;
       corr_wrdata_req_r7_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r7_reg <= local_wdata_req;
       end
       end //write_state 4'd7
       4'd8: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 9;
       reset_address <= 1'b1;
       corr_wrdata_req_r8_reg <= 1'b0;
end
```

```
      end
else if(out_stream_ready & ~out_stream_closed)
      begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r8_reg <= local_wdata_req;
      end
      end //write_state 4'd8
      4'd9: begin
if (reached_data_out_count)
      begin
        write_req <= 1'b0;
        if (writes_remaining == 0)
begin
      reached_data_out_count <= 1'b0;
      write_state <= 10;
      reset_address <= 1'b1;
      corr_wrdata_req_r9_reg <= 1'b0;
end
      end
else if(out_stream_ready & ~out_stream_closed)
      begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r9_reg <= local_wdata_req;
      end
      end //write_state 4'd9
      4'd10: begin
if (reached_data_out_count)
      begin
        write_req <= 1'b0;
        if (writes_remaining == 0)
begin
      reached_data_out_count <= 1'b0;
      write_state <= 11;
      reset_address <= 1'b1;
      corr_wrdata_req_r10_reg <= 1'b0;
end
      end
else if(out_stream_ready & ~out_stream_closed)
      begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r10_reg <= local_wdata_req;
      end
      end //write_state 4'd10
      4'd11: begin
if (reached_data_out_count)
      begin
        write_req <= 1'b0;
        if (writes_remaining == 0)
begin
      reached_data_out_count <= 1'b0;
      write_state <= 12;
      reset_address <= 1'b1;
      corr_wrdata_req_r11_reg <= 1'b0;
end
      end
else if(out_stream_ready & ~out_stream_closed)
      begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r11_reg <= local_wdata_req;
      end
      end //write_state 4'd11
      4'd12: begin
if (reached_data_out_count)
      begin
        write_req <= 1'b0;
        if (writes_remaining == 0)
begin
```

```
       reached_data_out_count <= 1'b0;
       write_state <= 13;
       reset_address <= 1'b1;
       corr_wrdata_req_r12_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r12_reg <= local_wdata_req;
       end
       end //write_state 4'd12
       4'd13: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       reached_data_out_count <= 1'b0;
       write_state <= 14;
       reset_address <= 1'b1;
       corr_wrdata_req_r13_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r13_reg <= local_wdata_req;
       end
       end //write_state 4'd13
       4'd14: begin
if (reached_data_out_count)
       begin
         write_req <= 1'b0;
         if (writes_remaining == 0)
begin
       state <= 4;
       reset_address <= 1'b1;
       corr_wrdata_req_r14_reg <= 1'b0;
end
       end
else if(out_stream_ready & ~out_stream_closed)
       begin
if (local_ready)
write_req <= 1'b1;
corr_wrdata_req_r14_reg <= local_wdata_req;
       end
       end //write_state 4'd14
endcase
              end // 4'd1

              4'd4: begin
                      if (writes_remaining == 0)
                        begin
//                          state <= 0;
  test_complete <= 1'b1;
                        end
                    end // 4'd2

              4'd1: begin
case (read_state)
       4'd0: begin
read_state <= 1;
       end
       4'd1: begin
if(in_stream_x1_ready && in_stream_x2_ready && in_stream_x3_ready &&
in_stream_x4_ready && in_stream_x5_ready && in_stream_x6_ready) begin
       if (local_ready && ~read_done) begin
read_req <= 1'b1;
```

```
read_done <= 1'b1;
read_state  <= 1;
stream_x1_wr_enable_reg <= 1'b0;
stream_x2_wr_enable_reg <= 1'b0;
stream_x3_wr_enable_reg <= 1'b0;
stream_x4_wr_enable_reg <= 1'b0;
stream_x5_wr_enable_reg <= 1'b0;
stream_x6_wr_enable_reg <= 1'b0;
       end
       if(local_rdata_valid && read_done) begin
stream_x1_wr_enable_reg <= 1'b1;
stream_x2_wr_enable_reg <= 1'b1;
stream_x3_wr_enable_reg <= 1'b1;
stream_x4_wr_enable_reg <= 1'b1;
stream_x5_wr_enable_reg <= 1'b1;
stream_x6_wr_enable_reg <= 1'b1;
read_req <= 1'b0;
read_done <= 1'b0;
read_state  <= 1;
       end
end
       end //read_state 4'd1
       4'd3: begin
read_req <= 1'b0;//just in case
reset_address <= 1'b1;
       end //read_state 4'd3
endcase
if (reached_data_count) begin
       read_state <= 7;
       state <= 3;
       read_req <= 1'b0;
       reset_address <= 1'b1;
end
              end // 4'd3
          endcase // state

          if (reset_address)
            begin
              //(others => '0')
              cs_addr <= MIN_CHIPSEL;

              row_addr <= 0;
              bank_addr <= 0;
              col_addr <= 0;
            end
          else if ((local_ready && read_req) && (state == 1))
                    begin
                      read_count <= read_count + 1'b1;
              if (col_addr >= DATA_COL)
                begin
                  col_addr <= 0;
                  if (row_addr == DATA_ROW)
                    begin
                      row_addr <= 0;
                      if (bank_addr == DATA_BANK)
                        begin
                          bank_addr <= 0;
                          if (cs_addr == DATA_CHIPSEL)
                              //reached_max_count <= TRUE
                              //(others => '0')
                              cs_addr <= MIN_CHIPSEL;

                          else
                            cs_addr <= cs_addr + 1'b1;
                        end
                      else
                        bank_addr <= bank_addr + 1'b1;
                    end
                  else
                    row_addr <= row_addr + 1'b1;
                end
```

```
            else
              col_addr <= col_addr + (2 * 2);
                  end
             else if ((local_ready && write_req) && (state == 3))
            if (col_addr >= DATA_OUT_COL)
              begin
                col_addr <= 0;
                if (row_addr == DATA_OUT_ROW)
                  begin
                    row_addr <= 0;
                    if (bank_addr == DATA_OUT_BANK)
                      begin
                        bank_addr <= 0;
                        if (cs_addr == DATA_OUT_CHIPSEL)
                            //reached_max_count <= TRUE
                            //(others => '0')
                            cs_addr <= MIN_CHIPSEL;

                        else
                          cs_addr <= cs_addr + 1'b1;
                      end
                    else
                      bank_addr <= bank_addr + 1'b1;
                  end
                else
                  row_addr <= row_addr + 1'b1;
              end
            else
              col_addr <= col_addr + (2 * 2);
           end
     end



//-------------------------------------------------------------
//LFSR re-load data storage
//Comparator masking and test pass signal generation
//-------------------------------------------------------------
always @(posedge clk or negedge reset_n)
  begin
    if (reset_n == 0)
      begin
        dgen_ldata <= 0;
        last_wdata_req <= 1'b0;
        //all ones
        compare_valid <= -1;

        //all ones
        compare_valid_reg <= -1;

        pnf_persist <= 1'b1;
        pnf_persist1 <= 1'b1;
        //all ones
        compare_reg <= -1;

        last_rdata_valid <= 1'b0;
      end
    else
      begin
        last_wdata_req <= wdata_req;
        last_rdata_valid <= local_rdata_valid;
        compare_reg <= compare;
        if (wdata_req)
            //Store the data from the first write in a burst
            //Used to reload the lfsr for the first read in a burst in WRITE
                                                              1, READ 1 mode

            if (wait_first_write_data)
                dgen_ldata <= dgen_data;
        //Enable the comparator result when read data is valid
        if (last_rdata_valid)
```

```
            compare_valid <= compare_reg;
          //Create the overall persistent passnotfail output
          if (~&compare_valid)
              pnf_persist1 <= 1'b0;
          //Extra register stage to help Tco / Fmax on comparator output pins
          compare_valid_reg <= compare_valid;

          pnf_persist <= pnf_persist1;
        end
    end


endmodule
```

Appendix A9. Software Control Application

altera_diag.c

```
// Code was created by Maxim Leonov
// Copyright 2008 Auckland University of Technology

#include "../lib/altera_lib.h"
#include "samples/shared/pci_diag_lib.h"
#include <stdio.h>

#include <time.h>
#include <stdlib.h>
#include <windows.h>
#include <winbase.h>

// input of command from user
static char line[256];

char *ALTERA_GetAddrSpaceName(ALTERA_ADDR addrSpace)
{
    return
        addrSpace==ALTERA_AD_BAR0 ? "Addr Space BAR0" :
        addrSpace==ALTERA_AD_BAR1 ? "Addr Space BAR1" :
        addrSpace==ALTERA_AD_BAR2 ? "Addr Space BAR2" :
        addrSpace==ALTERA_AD_BAR3 ? "Addr Space BAR3" :
        addrSpace==ALTERA_AD_BAR4 ? "Addr Space BAR4" :
        addrSpace==ALTERA_AD_BAR5 ? "Addr Space BAR5" :
        "Invalid";
}

void ALTERA_AccessRanges(ALTERA_HANDLE hALTERA)
{
    int cmd, cmd2;
    int i;
    UINT32 addr, data;
    ALTERA_ADDR ad_sp = ALTERA_AD_BAR0;
    ALTERA_MODE ad_mode = ALTERA_MODE_DWORD;

    for (i = ALTERA_AD_BAR0;
        i<ALTERA_ITEMS && !ALTERA_IsAddrSpaceActive(hALTERA, (ALTERA_ADDR)i);
                                                                    i++)
    {}
    ad_sp = (ALTERA_ADDR)i;
    if (ad_sp==ALTERA_ITEMS)
    {
        printf ("No active memory or IO ranges on board!\n");
        return;
    }

    do
    {
        printf ("Access the board's memory and IO ranges\n");
        printf ("---------------------------------------\n");
        printf ("1. Change active memory space:
                        %s\n",ALTERA_GetAddrSpaceName(ad_sp));
        printf ("2. Toggle active mode: %s\n",
            ad_mode==ALTERA_MODE_BYTE ? "BYTE (8 bit)" :
            ad_mode==ALTERA_MODE_WORD ? "WORD (16 bit)" : "DWORD (32 bit)");
        printf ("3. Read from board\n");
        printf ("4. Write to board\n");
             printf ("99. Back to main menu\n");
        printf ("\n");
        printf ("Enter option: ");
        cmd = 0;
```

```
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d",&cmd);
        switch (cmd)
        {
        case 1:
            printf ("Choose memory or IO space:\n");
            printf ("--------------------------\n");
            for (i=ALTERA_AD_BAR0; i<ALTERA_ITEMS; i++)
            {
                printf ("%d. %s", i+1,
                    ALTERA_GetAddrSpaceName((ALTERA_ADDR)i));
                if (!ALTERA_IsAddrSpaceActive(hALTERA, (ALTERA_ADDR)i))
                    printf (" - space not active");
                printf("\n");
            }
            printf ("Enter option: ");
            cmd2 = 99;
            fgets(line, sizeof(line), stdin);
            sscanf(line, "%d",&cmd2);
            if (cmd2>=1 && cmd2<ALTERA_ITEMS+1)
            {
                ad_sp = (ALTERA_ADDR)(cmd2-1);
                if (!ALTERA_IsAddrSpaceActive(hALTERA, ad_sp))
                    printf ("Chosen space not active!\n");
            }
            break;
        case 2:
            ad_mode = (ALTERA_MODE)((ad_mode + 1) % 3);
            break;
        case 3:
            printf ("Enter offset to read from: ");
            fgets(line, sizeof(line), stdin);
            sscanf (line, "%x", &addr);
            switch (ad_mode)
            {
            case ALTERA_MODE_BYTE:
                data = ALTERA_ReadByte(hALTERA, ad_sp, addr);
                break;
            case ALTERA_MODE_WORD:
                data = ALTERA_ReadWord(hALTERA, ad_sp, addr);
                break;
            case ALTERA_MODE_DWORD:
                data = ALTERA_ReadDword(hALTERA, ad_sp, addr);
                break;
            }
            printf ("Value read: %x\n", data);
            break;
        case 4:
            printf ("Enter offset to write to: ");
            fgets(line, sizeof(line), stdin);
            sscanf (line,"%x", &addr);
            printf ("Enter data to write %s: ",
                ad_mode==ALTERA_MODE_BYTE ? "BYTE (8 bit)" :
                ad_mode==ALTERA_MODE_WORD ? "WORD (16 bit)" : "DWORD (32
                                                                bit)");
            fgets(line, sizeof(line), stdin);
            sscanf (line, "%x",&data);
            switch (ad_mode)
            {
            case ALTERA_MODE_BYTE:
                ALTERA_WriteByte(hALTERA, ad_sp, addr, (BYTE) data);
                break;
            case ALTERA_MODE_WORD:
                ALTERA_WriteWord(hALTERA, ad_sp, addr, (WORD) data);
                break;
            case ALTERA_MODE_DWORD:
                ALTERA_WriteDword(hALTERA, ad_sp, addr, data);
                break;
            }
            break;
        }
```

```
    } while (cmd!=99);
}

void DLLCALLCONV ALTERA_IntHandlerRoutine(ALTERA_HANDLE hALTERA,
ALTERA_INT_RESULT *intResult)
{
    printf ("Got Int number %ld\n", intResult->dwCounter);
}

void ALTERA_EnableDisableInterrupts(ALTERA_HANDLE hALTERA)
{
    int cmd;

    printf ("WARNING!!!\n");
    printf ("----------\n");
    printf ("Your hardware has level sensitive interrupts.\n");
    printf ("You must modify the source code of ALTERA_IntEnable(), in the
                                                file altera_lib.c,\n");
    printf ("to acknowledge the interrupt before enabling interrupts.\n");
    printf ("Without this modification, your PC will HANG upon interrupt!\n");
    printf ("\n");

    do
    {
        printf ("Enable / Disable interrupts\n");
        printf ("---------------------------\n");
        printf ("1. %s Int\n", ALTERA_IntIsEnabled(hALTERA) ? "Disable" :
                                                            "Enable");
        printf ("99. Back to main menu\n");
        printf ("\n");
        printf ("Enter option: ");
        cmd = 0;
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d",&cmd);
        switch (cmd)
        {
        case 1:
            if (ALTERA_IntIsEnabled(hALTERA))
            {
                printf ("Disabling interrupt Int\n");
                ALTERA_IntDisable(hALTERA);
            }
            else
            {
                printf ("Enabling interrupt Int\n");
                if (!ALTERA_IntEnable(hALTERA, ALTERA_IntHandlerRoutine))
                    printf ("failed enabling interrupt Int\n");
            }
            break;
        }
    } while (cmd!=99);
}

ALTERA_HANDLE ALTERA_LocateAndOpenBoard (DWORD dwVendorID, DWORD dwDeviceID)
{
    DWORD cards, my_card;
    ALTERA_HANDLE hALTERA = NULL;

    if (dwVendorID==0)
    {
        printf("Enter VendorID: ");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%lx", &dwVendorID);

        printf("Enter DeviceID: ");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%lx", &dwDeviceID);
    }
    cards = ALTERA_CountCards (dwVendorID, dwDeviceID);
    if (cards==0)
    {
```

```
        printf("%s", ALTERA_ErrorString);
        return NULL;
    }
    else if (cards==1) my_card = 1;
    else
    {
        DWORD i;

        printf("Found %ld matching PCI cards\n", cards);
        printf("Select card (1-%ld): ", cards);
        i = 0;
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%ld",&i);
        if (i>=1 && i <=cards) my_card = i;
        else
        {
            printf ("Choice out of range\n");
            return NULL;
        }
    }
    if (ALTERA_Open (&hALTERA, dwVendorID, dwDeviceID, my_card - 1))
        printf ("ALTERA PCI card found!\n");
    else printf ("%s", ALTERA_ErrorString);
    return hALTERA;
}

int main()
{
    int cmd, j, i;
       int fileValue1, fileValue2;
       char c;
       UINT32 data, k;
    ALTERA_HANDLE hALTERA = NULL;
    HANDLE hWD;
    ALTERA_ADDR ad_sp = ALTERA_AD_BAR0;
    ALTERA_MODE ad_mode = ALTERA_MODE_DWORD;
       INT32 file1[sequences_length];
       INT32 file2[sequences_length];
       FILE *f1;
       FILE *f2;
       FILE *res;
       LARGE_INTEGER ticksPerSecond;
       LARGE_INTEGER tick;   // A point in time
       LARGE_INTEGER start_ticks, end_ticks, cputime;



    printf ("Software Control Application.\n");
    printf ("Application accesses hardware using " WD_PROD_NAME ".\n");

    // make sure WinDriver is loaded
    if (!PCI_Get_WD_handle(&hWD)) return 0;
    WD_Close (hWD);

    if (ALTERA_DEFAULT_VENDOR_ID)
        hALTERA = ALTERA_LocateAndOpenBoard(ALTERA_DEFAULT_VENDOR_ID,
ALTERA_DEFAULT_DEVICE_ID);

    do
    {
        printf ("\n");
        printf ("CORRELATOR main menu\n");
        printf ("--------------------\n");
        printf ("1. Scan PCI bus\n");
        printf ("2. Locate/Choose ALTERA board\n");
        if (hALTERA)
        {
            printf ("3. PCI configuration registers\n");
            printf ("4. Access ALTERA memory and IO ranges\n");
            printf ("5. Enable / Disable interrupts\n");
                    printf ("6. Write data for correlator\n");
```

```
        }
        printf ("99. Exit\n");
        printf ("Enter option: ");
        cmd = 0;
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d",&cmd);
        switch (cmd)
        {
        case 1: // Scan PCI bus
            PCI_Print_all_cards_info();
            break;
        case 2: // Locate ALTERA board
            if (hALTERA) ALTERA_Close(hALTERA);
            hALTERA = ALTERA_LocateAndOpenBoard(0, 0);
            if (!hALTERA) printf ("ALTERA card open failed!\n");
            break;
        case 3: // PCI configuration registers
            if (hALTERA)
            {
                WD_PCI_SLOT pciSlot;
                ALTERA_GetPciSlot(hALTERA, &pciSlot);
                PCI_EditConfigReg(pciSlot);
                break;
            }
        case 4: // Access ALTERA memory and IO ranges
            if (hALTERA) ALTERA_AccessRanges(hALTERA);
            break;
        case 5: // Enable / Disable interrupts
            if (hALTERA)
                ALTERA_EnableDisableInterrupts(hALTERA);
            break;
        case 6:
                    f1 = fopen("signalA_in.dat", "r");
                    if ( f1 == NULL ) {
                           fprintf(stderr, "Error opening input file %s\n",
"signalA_in.dat");
                    }

                    f2 = fopen("signalB_in.dat", "r");
                    if ( f2 == NULL ) {
                           fprintf(stderr, "Error opening input file %s\n",
"signalB_in.dat");
                    }
                    res = fopen("results.txt", "w");

                    i = 0;
                    while (fscanf(f1,"%d",&fileValue1) != EOF) {
                           file1[i] = fileValue1;
                           i++;
                    }
                    i = 0;
                    while (fscanf(f2,"%d",&fileValue2) != EOF) {
                           file2[i] = fileValue2;
                           i++;
                    }

                    printf ("\tValue of CLOCKS_PER_SEC is :    %i
ticks/sec\n",CLOCKS_PER_SEC   );
                    // get the high resolution counter's accuracy
                    if (!QueryPerformanceFrequency(&ticksPerSecond))
                           printf("\tno go QueryPerformance not present");
                    printf ("\tfreq test:   %I64Ld ticks/sec\n",ticksPerSecond
);
                    // what time is it?
                    if (!QueryPerformanceCounter(&tick) ) printf("no go
counter not installed");
                           printf ("\tQueryPerformanceCounter testpoint :
%I64Ld  ticks\n",tick);
                    QueryPerformanceCounter(&start_ticks);
                    /* start foo()  */
                    printf ("\t\t\tWriting data for correlator...\n");
```

```
                    for(i = 0, j = 0; i <= sequences_length; i++, j = j+ 4)
//writing with raw data
                    {
                           ALTERA_WriteWord(hALTERA, ad_sp, j, (WORD)file1[i]);
                           ALTERA_WriteWord(hALTERA, ad_sp, j+2,
                                                        (WORD)file2[i]);
                    /*     ALTERA_WriteWord(hALTERA, ad_sp, j, (WORD)0);
                           ALTERA_WriteWord(hALTERA, ad_sp, j+2, (WORD)1);*/
                    }
                    printf ("\t\t\tFinished writing data for correlator.\n");
                    printf ("\t\tEnabling correlator design.\n");
                    ALTERA_WriteDword(hALTERA, ALTERA_AD_BAR2, 0xC, 0x400);
                                                 //triggering correlator
                    printf ("\t\t\tWaiting for completion...\n");
                    do
                    {
                           data = ALTERA_ReadDword(hALTERA, ALTERA_AD_BAR2,
                                                                    0xC);
                           //printf("%h", data);

                    }while(data == 0x80000);
                    printf("\t\tReading of processed data completed.\n");

                    /* end foo( ) */
                    QueryPerformanceCounter(&end_ticks);
                    cputime.QuadPart = end_ticks.QuadPart-
                                                 start_ticks.QuadPart;
                    printf ("\tElapsed CPU time test:   %.9f  sec  ticks
                                                        %d\n",
                    ((float)cputime.QuadPart/(float)ticksPerSecond.QuadPart),
                                                    cputime.QuadPart);

                    printf("\t\tProcessing completed.\n");
                    printf ("\t\t\tReading processed data back...\n");
                    for(k = 0; k <= 64; k = k + 4) //reading processed data
                                                 back
                    {
                           data = ALTERA_ReadDword(hALTERA, ad_sp, k);
                           fprintf(res, "%i %i\n", k, data);
                           //printf ("Value read: %x\n", data);
                    }
                    fclose(f1);
                    fclose(f2);
                    fclose(res);
                    break;

        }
    } while (cmd!=99);

    if (hALTERA) ALTERA_Close(hALTERA);

    return 0;
}
```